SML# Document
Version 4.2.0

Atsushi Ohori Katsuhiro Ueno
The SML# development team

Mar, 2025

Contents

Part I

Overview

Chapter 1

Preface

This is the official document of a functional programming language SML#. This document intends to provide comprehensive information on SML#, including: tutorials on SML# programming, reference manuals of the SML# language and the basis library, and, detailed descriptions of the internals and data structures of the SML# system.

Send comments and questions to the authors.

June, 2017

Atsushi Ohori Katsuhiro Ueno
The SML# development team

Chapter 2

About This Document

This documents consist of the following parts.

1. Part I: an overview of the SML#.
2. Part II: SML# tutorials. The tutorials are structured in such a way that the reader can learn SML# from programming basics to advanced programming features provided by SML#. We recommend to read through this part quickly.

   If you are in a harry, you may find the desired information in one of the following pages.

   - **How to install SML#**: Chapter 5.
   - `smlsharp` **command parameters**: Section 6.5.
   - **SML# development team and contact information**: Section 3.3.
   - **On the name of SML#**: Section 3.2 (History of SML#).
   - **SML# web page**: `https://smlsharp.github.io/`
   - **SML# document in Japanese**: `https://smlsharp.github.io/ja/documents/4.2.0/`
3. Part III: reference manuals on the SML# language and the libraries including Standard ML Basis Library.
4. Part IV: Reference manuals on SML# tools (`smllex`, `smlyacc`, `smlformat` etc).
5. Part V: The internals and data structures of the SML# system.
6. Part VI: references.

In the version 4.2.0 of this document, Part IV and **??** are not completed. They are going to be completed in the future version.

Chapter 3

Overview of SML#

This chapter outlines the SML# language and the compiler.

3.1 What is SML# ?

SML# is a new programming language in the ML-family, having the following features.

1. **Backward compatibility with Standard ML.** SML# can compile all programs that conform to the definition of the Standard ML[5] with a few exceptions.

2. **Record polymorphism.** SML# supports record polymorphism [9]. In SML# , field selection operators `#` ⟨*label*⟩ and record patterns { ⟨*field-pat*⟩ , . . . } are fully polymorphic. This feature is essential in modular development of programs manipulating records, and is the key to extend ML with SQL.

3. **Seamless integration of SQL.** SML# seamlessly integrates (currently a subset of) SQL. Instead of providing built-in primitives to access database servers, SML# integrate SQL expressions themselves as polymorphically-typed first-class citizens. This allows the programmer to directly access databases in your polymorphic ML code.

4. **Direct interface to C.** SML# programs can directly call C functions of your own coding or in system libraries. The programmer need only declare their names and types, without writing mysterious "stubs" or conversion functions. The SML# generates external references to C functions, which are resolved and linked by the system linker. Both static and dynamic linking are supported.

5. **Separate compilation and linking.** SML# supports true separate compilation and linking. By writing an interface file, each source file is compiled separately into an object file in the standard system format (e.g. ELF format.) The separately compiled object files are then linked together possibly with C functions and libraries into an executable program.

6. **Multithread support for multicore CPUs.** The non-moving GC [16] and direct C interface allow SML# code to directly call POSIX thread library. As far as the OS thread library support a multicore CPU, SML# program automatically obtains multithread capability for multicore CPUs. With the non-moving fully concurrent GC we have just developed, concurrent threads run efficiently on a multicore CPU.

The SML# compiler and its runtime system are developed by the SML# development team. They are open-source software distributed under the MIT license copyrighted by the SML# development team (See Section 4). The current SML# development team consists of Atsushi Ohori (Tohoku University) and Katsuhiro Ueno (Niigata University) (in the alphabetical order).

3.2 History of SML#

The origin of the SML# can be traced back to the proposal of Machiavelli published in [11]. This paper showed that ML with record polymorphism can be extended with SQL and reported a implementation of a prototype interpreter.

In 1993, Atsushi Ohori extended the Standard ML of New Jersey compiler at Kansai Laboratory of Oki Electric, and named the experimental prototype **SML# of Kansai**. The Internet still remembers the posting of **SML# of Kansai** to the types mailing list (`http://www.funet.fi/pub/languages/ml/sml%23/description`).

The name **SML# of Kansai** symbolized the field selector `#`⟨*label*⟩ , which was given a polymorphic type for the first time by this compiler. This compiler was reported in the ACM TOPLAS article on record polymorphism [9] as **SML#**.

To support not only record polymorphism but also inseparability and other practically important features, we decided to develop a new SML-style language from scratch, and in 2003, we started the SML# compiler project at Japan Advanced Institute of Science and Technology as a part of the e-Society project (`http://www.tkl.iis.u-tokyo.ac.jp/e-society/index.html`) funded by the Japan ministry of science, education and technologies

In 2006, the project moved to Tohoku University.

In 2021, the core activity of development moved to the SML# development team.

3.3 SML# Development Team and Contact Information

At present (2025-03-24), SML# is being developed by

- Atsushi Ohori （Professor emeritus, Tohoku University）
- Katsuhiro Ueno （Niigata University）

with help of students.

The past SML# development team members include (with the affiliation at the time of development):

- Atsushi Ohori (School of Information Science, JAIST; RIEC, Tohoku University)
- Kiyoshi Yamatodani (Sanpukoubou Inc)
- Nguyen Huu Duc (School of Information Science, JAIST; RIEC, Tohoku University)
- Liu Bochao (School of Information Science, JAIST; RIEC, Tohoku University)
- Satoshi Osaka (School of Information Science, JAIST)
- Katsuhiro Ueno (School of Information Science, JAIST; RIEC, Tohoku University; Academic Assembly Institute of Science and Technology, Niigata University)

Contact information regarding SML#:

- SML# home page: `https://smlsharp.github.io/`
- SML# forum: `https://github.com/smlsharp/forum/discussions`

  This board is for general discussion on SML#. To post a message, you need to have a GitHub account. All the messages are made available on the Web.
- SML# X (twitter) account: `@smlsharp`

  We tweet the latest information about SML# on this account.

  Send your questions, requests, and comments to the development team.

3.4 Acknowledgments

From its start in 2003, we have benefited from many peoples and organizations.

3.4.1 Project funding

SML# development was started as a part of the 5 year project“e-Society leading project: highly productive reliable software development technologies (Project Director: Professor Takuya Katayama)” under the title “ reliable software development technology based on automatic program analysis (chief investigator: Atsushi Ohori) ” （`http://www.tkl.iis.u-tokyo.ac.jp/e-society/index.html`）sponsored by the Japan ministry of science, education and technologies.

3.4.2 Third-party code and software tool used in SML# development

Since the version 1.0 had completed, SML# has been developed by the SML# compiler itself. Before that, we had used Standard ML of New Jersey for development and MLTon compiler for building distributions.

The SML# compiler is a software developed by the SML# development team (3.3). We have wrote most of the code of SML# compiler from scratch, except for the following codes:

contents	location in SML# distribution	source code
ML-Yacc	src/ml-yacc	Standard ML of New Jersey 110.73
ML-Lex	src/ml-lex	Standard ML of New Jersey 110.73
SML/NJ Library	src/smlnj-lib	Standard ML of New Jersey 110.73
`TextIO`, `BinIO`, `OS`, `Date`, `Timer` structures	src/smlnj	Standard ML of New Jersey 110.73
floating point-string conversion (dtoa.c)	src/runtime/netlib	the Netlib

All of the above are open-source software that are compatible with SML# license. The SML# source distribution includes the license of each of them at the "location in SML# distribution" show above.

3.4.3 Collaborators

Many people have contributed to research and development of SML#. In addition to the development team (Section 3.3), the following people directly contributed to SML# research.

- Isao Sasano. With Atsushi Ohori, he investigated "Lightweight fusion by fixed point promotion" and developed an experimental inlining module that performs lightweight fusion. This feature is experimental and has not yet been integrated in SML# compiler, but we plan to adopt this method in a future version.

- Toshihiko Otomo. With Atsushi Ohori and Katsuhiro Ueno, he investigated the possibility of non-moving collector and showed an initial experimental result indicating that a non-moving GC is viable in functional languages.

Many other people helped us through collaborative research with Atsushi Ohori and others to develop type-theory and compilation methods that underlie SML# compiler. SML# compiler is directly based on the following research results, some of them were collaboratively done.

- record polymorphism [8, 9].

- database type inference [10].

- database language (Machiavelli)[11, 1].

- rank-1 polymorphism [15].

- unboxed semantics of ML [13].

- natural data representation for ML [7].

- lightweight fusion [12].

- efficient non-moving GC[16].

- implementation method for SQL integration[14].

- JSON support[17].

- fully concurrent GC[18].

We have also benefited from many other researchers from 1989. We refrain from compiling a comprehensive list, which seems to be impossible.

3.5 Limitations in SML# version 4.2.0

We have successfully developed all the features listed in Section 3.1, which include all the features that we had initially aimed at. The version 4.2.0 contains most of them, with the following restrictions.

1. **Target Architecture.**

   The current compiler only generates x86-64 and arm64 code and only supports Linux and macOS.

2. **Optimization.**

   In this version, the compiler is equipped with standard optimizations such as inlining and constant propagation. We hope that we will provide more advanced optimizations, which will make the compiler as fast as other mature compilers.

3. **Limitations on arm64**.

   Arm64 support is still experimental. Some features do not work similarly to x86_64. For example, on arm64 macOS, exporting an ML function to C sometimes causes the following LLVM error:

   ```
   LLVM ERROR: INIT_TRAMPOLINE operation is only supported on Linux
   ```

   This limitation will be resolved in the future versions.

Chapter 4

SML# License

Part II

Tutorials

Chapter 5

Installing SML#

5.1 Prerequisites

The part II provides tutorials to get started writing SML# programs. Let us begin by installing the SML# compiler and setting up a programming environment.

The SML# version 4.2.0 works on one of the following platforms.

- Linux (amd64 (x86_64) or aarch64 (arm64))
- macOS (version 10.15 or later, x86_64 or arm64)
- Windows 11 (Windows Subsystem for Linux 2)

SML# compiler requires the following software.

- MassiveThreads 1.02
- GNU Multiple Precision Arithmetic (GMP) library
- LLVM (from 7.1.0 to 20.1; the latest one available is preferrable)

MassiveThreads is a free software distributed under a BSD-style license. GMP is a free software distributed under LGPL (GNU Lesser General Public License). LLVM is a open-source software distributed under Apache license version 2 with some exceptions.

LLVM version must be in version 7.1.0 or above. The latest LLVM available in your system is preferable. We confirmed that SML# works with up to LLVM 20.1. Compliing SML# with LLVM prior to 7 shall fail.

All of these libraries and commands are required not only to build SML# but also to run the SML# compiler command and SML# programs. Note the following:

- These libraries and commands are used to build SML#. The MassiveThreads and GMP libraries are liniked into the SML# compiler command. Libraries of LLVM is not linked into the SML# compiler.
- After installation, the SML# compiler command uses `opt`, `llc`, `llvm-as`, and `llvm-dis` command for code generation.
- The executable file of any user program compiled by the SML# compiler is linked with the MassiveThreads and GMP library, even if the program does not use the features of these libraries at all. We plan to fix this in upcoming releases.

Since these libraries and commands are not included in the SML# distribution package, you need to install them before installing SML#. In most cases, these can be easily installed by the package management system you use.

Below, we show the details of SML# installation steps for each of the supported operating systems.

5.2 Debian GNU/Linux

Private repositories of SML# are provided for the latest stable version (when SML# is released) of Debian GNU/Linux and Debian sid. Add one of these repositories to your system, and you can install and update SML# by `apt` command.

Do the following commmands to set up the SML# compiler.

- Debian sid:

```
wget -P /etc/apt/keyrings https://smlsharp.github.io/repos/debian/dists/sid/smlsharp-archive-k
wget -P /etc/apt/sources.list.d https://smlsharp.github.io/repos/debian/dists/sid/smlsharp.sou
apt update
apt install smlsharp
```

- Debian 12 (bookworm):

```
wget -P /etc/apt/keyrings https://smlsharp.github.io/repos/debian/dists/bookworm/smlsharp-archi
wget -P /etc/apt/sources.list.d https://smlsharp.github.io/repos/debian/dists/bookworm/smlshar
apt update
apt install smlsharp
```

- Debian 11 (bullseye):

```
wget -P /usr/share/keyrings https://smlsharp.github.io/repos/debian/dists/bullseye/smlsharp-ar
wget -P /etc/apt/sources.list.d https://smlsharp.github.io/repos/debian/dists/bullseye/smlshar
apt update
apt install smlsharp
```

We show some more details of installation below (for sid).

1. Download the public key of the SML# development team and put it to the specified position in the system.

   ```
   wget -P /etc/apt/keyrings https://smlsharp.github.io/repos/debian/dists/sid/smlsharp-archive-k
   ```

   You can check the fingerprint of the key by the following command:

   ```
   gpg --with-fingerprint /etc/apt/keyrings/smlsharp-archive-keyring.gpg
   ```

   Confirm that the fingerprint of the downloaded key is identical to the above fingerprint (those who would like to verify the public key strictly should meet a developer of the SML# compiler and receive the fingerprint of the key). The fingerprint of the key is the following:

   ```
   DD99 2B50 C9A3 B075 DA04 613A D299 F71F C5C1 D12E
   ```

2. Download the description file of the private repository and add it to the system.

   ```
   wget -P /etc/apt/sources.list.d https://smlsharp.github.io/repos/debian/dists/sid/smlsharp.sou
   ```

3. Obtain the package list from the repository.

   ```
   apt update
   ```

4. Install the SML# compiler. Prerequisite libraries, such as LLVM and MassiveThreads, are installed if needed. SML# tools, such as SMLFormat, are also installed as recommended packages.

   ```
   apt install smlsharp
   ```

5.3 Ubuntu

The private repositories are provided for Ubuntu LTS releases avaliable when SML# is released through PPA (Personal Package Archives for Ubuntu). Add one of these repositories to your system, and you can install and update the SML# compiler by the `apt` command.

Do the following commmands to set up the SML# compiler.

- 22.04 LTS (jammy), 24.04 LTS (noble):

```
add-apt-repository ppa:smlsharp/ppa
apt update
apt install smlsharp
```

`add-apt-repository` adds the private repository to the system. The `apt` commands are similar to those for Debian. See Section 5.2 for details.

5.4 Fedora

Private repositories of SML# are provided for the latest version (when SML# is released) of Fedora Rawhide. Add one of these repositories to your system, and you can install and update SML# by `dnf` command.

Do the following commands to set up the SML# compiler.

- Fedora Rawhide:

```
rpm -i https://smlsharp.github.io/repos/fedora/smlsharp-release-fedora-41-1.noarch.rpm
dnf install smlsharp smlsharp-smlformat smlsharp-smllex smlsharp-smlyacc
```

We show some more details of installation below (for Rawhide).

1. Download and install the RPM package that includes the public key of the SML# development team and configuration file of the private repository.

   ```
   rpm -i https://smlsharp.github.io/repos/fedora/smlsharp-release-fedora-41-1.noarch.rpm
   ```

2. Install the SML# compiler and its related tools.

   ```
   dnf install smlsharp smlsharp-smlformat smlsharp-smllex smlsharp-smlyacc
   ```

   In the middle of `dnf` command execution, the command asks you several times for importing SML#'s public key. Check the fingerprint of the key and permit to import it. The fingerprint is given as follows:

   ```
   DD99 2B50 C9A3 B075 DA04 613A D299 F71F C5C1 D12E
   ```

5.5 AlmaLinux

Private repositories of SML# are provided for AlmaLinux 8 and 9. Add one of these repositoriers to your system, and you can install and update SML# by `yum` or `dnf` command.

Do the following commmands to set up the SML# compiler.

- AlmaLinux 8, AlmaLinux 9:

```
rpm -i https://smlsharp.github.io/repos/almalinux/smlsharp-release-almalinux-8-1.noarch.rpm
dnf install smlsharp smlsharp-smlformat smlsharp-smllex smlsharp-smlyacc
```

The commands are almost similar to those for Fedora. See Section 5.4 for details.

5.6 Nix/NixOS

Enable Flakes and run the following command to run the SML# compiler:

```
nix run github:smlsharp/nixpkgs
```

5.7 macOS

We prepare a Homebrew formula for SML#. After setting up Homebrew, invoke the following commands in order to install SML# and its dependent libraries.

```
brew tap smlsharp/smlsharp
brew install smlsharp
```

We show some more details below.

1. Consult `http://brew.sh/` and set up Homebrew.
2. Tap the git repository provided by the SML# development team so that the SML#-related pacakges are added to the Homebrew system.

   ```
   brew tap smlsharp/smlsharp
   ```

   This repository includes the formulae (packages) and bottles (binary packages) of the MassiveThread library and SML# compiler. Bottles are provided only for the latest version of macOS when SML# is released.
3. Install the SML# compiler by the following command.

   ```
   brew install smlsharp
   ```

   Prerequisite libraries, such as LLVM and MassiveThreads, are also automatically installed. If you use macOS for which the bottles are not provided, MassiveThreads and SML# are built from source and therefore the command takes a long time.

5.8 Windows 11

By setting up Windows Subsystems for Linux 2 and Ubuntu, you can install the SML#'s `.deb` package for Ubuntu in Windows.

See Microsoft's documents for setting up Windows Subsystems for Linux 2. After `bash` command becomes available, install SML# in the same installation steps as Ubuntu (see Section 5.3 for details).

5.9 Building from the source

For Linux and other systems, you need to build from the SML# source distribution. To do this, the following tools and libraries are required:

1. GNU binutils（GNU Binary Utilities),
2. C and C++ compiler (gcc or clang),
3. make (GNU make is recommended),
4. MassiveThreads and GMP library and their header files, and
5. LLVM library, its header files, and commands (supported versions are from 7.1.0 to 20.1).

If these have been already installed, you can build and install SML# in the following popular three steps: `./configure && make && make install`.

We recommend you to install these prerequisites through the packaging system of your OS. If your OS does not provide them, you need to build them from the source. See those official documents for details of this procedure.

For your information, we roughly present how to compile MassiveThreads and LLVM 7.1.0 when we write this document.

MassiveThreads Obtain the source code of MassiveThreads 1.02 named `massivethreads-1.02.tar.gz` from MassiveThreads web site `https://github.com/massivethreads/massivethreads`. After expanding the tar archive, do `./configure && make && make install`.

LLVM 7.1.0 Obtain the source code of LLVM 7.1.0 named `llvm-7.1.0.tar.xz` from LLVM web site `http://llvm.org/`. After expanding the tar archive, do the following five commands:

```
mkdir build
cd build
cmake -G "Unix Makefiles" \
      -DCMAKE_INSTALL_PREFIX=/where/llvm/is \
      -DCMAKE_BUILD_TYPE=Release \
      -DLLVM_BUILD_LLVM_DYLIB=On \
      -DLLVM_ENABLE_BINDINGS=Off \
      ..
make
make install
```

The above options specified to `configure` are optional but suggested. `-DCMAKE_INSTALL_PREFIX` option should be specified to avoid conflict with other installation of LLVM. `-DCMAKE_BUILD_TYPE=Release` option enforces compiling LLVM libraries by an optimizing compiler. `-DLLVM_BUILD_LLVM_DYLIB=On` option enforces building the LLVM shared library. `-DLLVM_ENABLE_BINDINGS=Off` option avoids to build modules unnecessary for SML#.

With these preparation, SML# can be build in the following steps.

1. Download the source distribution from: `https://github.com/smlsharp/smlsharp/releases/download/v4.2.0/smlsharp-4.2.0.tar.gz`. The latest version of the source package is also available from `https://smlsharp.github.io/ja/downloads/`.

2. Select an SML# source directory and extract the tar archive there. Note for non-English users: the source directory must not include non-ASCII characters, otherwise the build process will fail.

3. Select an SML# installation destination directory. Let *prefix* be the path to the directory.

4. In the SML# source directory, execute `configure` script.

   ```
   $ ./configure --prefix=prefix --with-llvm=/where/llvm/is
   ```

   You can specify `--prefix=`*prefix* option to specify the destination directory. If `--prefix` option is omitted, `/usr/local` is used as the destination directory. If `--with-llvm` option is omitted, the `configure` script searches for LLVM libraries and commands from the standard directories such as `/usr/bin`.

5. Do `make` command.

   ```
   $ make
   ```

   After `make` is finished, you can launch the SML# compiler without installing it by the following commands.

   ```
   $ src/compiler/smlsharp -Bsrc
   ```

6. (Optional) The following commands build the SML# compiler by the SML# compiler you built in the previous step.

   ```
   $ make stage
   $ make
   ```

7. Do `make install` command.

   ```
   $ make install
   ```

   If you want to put files to be installed in a directory *prefix*′ different from *prefix*, specify `DESTDIR=`*prefix*′ option to `make install` command.

The following files are installed by the above procedure.

1. `smlsharp` command at *prefix*`/bin/smlsharp`
2. `smlformat` command at *prefix*`/bin/smlformat`
3. `smllex` command at *prefix*`/bin/smllex`
4. `smlyacc` command at *prefix*`/bin/smlyacc`
5. library files in the *prefix*`/lib/smlsharp/` directory.

If successful, you can invoke SML# by typing:

```
$ prefix/bin/smlsharp
```

Some hints:

- This process compiles all the source files including those of tools, which takes some time. If you have a CPU with n cores and use GNU make, then try to give `-j`m ($m \leq n$) switch to `make` command for parallel processing, where m indicate the degree of parallelism.
- In addition to LLVM for the SML# compiler to work, SML# compiler developers need to install LLVM 7.1.0 as well for bootstrapping. If you would like to develop the SML# compiler, add `--with-llvm7` option to `./configure` command line to specify where LLVM 7.1.0 is installed.

Chapter 6

Setting up SML# programming environment

6.1 Unix-family OS, Emacs, and other tools

To enjoy writing programs, you need to set up a programming environment including

- a good text editor, and
- a compiler and a linker.

The environment required for SML# programming is essentially the same as in any other programming, except of course for the SML# compiler. In Java and other languages, an integrated development environment such as Eclipse is often used, but for SML# programming, we recommend the following standard system development environments.

- **OS in the Unix family.** A Unix-family OS such as Linux, FreeBSD (including Mac OS) provides a rich collection of programming tools. This would be your first choice. It is easy to set up Linux on Windows using a virtual machine such as VMWare.

  reasonable alternative.

- **Emacs editor.** This is one of the best editor for programming, which is a repeated process of editing an ASCII text file and and compiling it. In most of the time, you are interacting with your text editor. So choosing a highly custormisable high-performance editor is important. Among various choices, we recommend one in the Emacs-family (GNU Emacs, XEmacs). Emacs is a powerful custormisable text editor, and it can also perform command execution and file system management. It requires some practice at the beginnings, but once you mastered its basic functionality, it will become a powerful tool in programming.

- **C compiler.** SML# compiler generates x86_64 or arm64 native code, creates an object file in a standard format (e.g. ELF), and generates an executable code by linking object files with C libraries. In this process, it calls C compiler driver command such as `gcc` or `clang`. One of them should already be installed in a Unix-family OS.

  If your purpose is to make a small program entirely within SML#, then you will not need to invoke a C compiler directly. However, if you want to make a practical program, you may want to call some system library functions or you may write some part of your system in C and call that function from your ML code. This is straightforward in SML#. To exploit this feature, we recommend that you familiarize yourself with C compiler.

- **database systems.** SML# seamlessly integrates SQL. SML# version 4.2.0 supports PostgreSQL, MySQL, ODBC, and SQLite3. If you set up one of them, then you can use a database system directly within your SML# code.

6.2 Bootstrapping the SML# compiler

This section outlines the structure of SML# compiler and the method to building (bootstrapping) it. You do not have to understand this section for installing and using SML# compiler, but you may find this section informative in understanding various messages during compilation of SML# and also a structure of a compiler in general.

SML# system consists of a single compiler that performs separate compilation. Its interactive mode is realized by the top-level-loop performing the following steps:

1. compile the user input using the current static environment as its interface,
2. link the object file with the current system to generate a shared executable file,
3. dynamically load the shared executable in the current system, and call its entry point.

SML# is written in SML#, C, and C++. In addition, it uses the following tools during compiling the SML# compiler.

- ml-lex, ml-yacc: a lexical analyzer generator and a parser generator.
- SMLFormat: a printer generator.
- The Standard ML Basis Library.

All of them are written in Standard ML.

SML# compiler compiles each SML# (which is a super set of Standard ML) source file (`source.sml`) into a system standard object file (`sample.o`). To generate an executable file, the compiled files are then linked by the standard linker (`ld` in Unix-family OS) invoked through C/C++ compiler driver command (`gcc` or `clang`). So, in order to build the SML# compiler, it is sufficient to have a C/C++ compiler and an SML# compiler. But of course, at the time when the SML# compiler is first built, an SML# compiler is not available. The standard step of solving this bootstrap problem is the following.

1. Compile SML# runtime library written in C/C++ and archive it as a static link library.
2. Obtain a pre-compiled LLVM IR source file of a minimal SML# compiler `minismlsharp` that is sufficient for compiling all the source files used in the SML# compiler. The pre-compiled files are typically generated by an older version of SML# compiler.
3. In the system where the target SML# compiler is installed, assemble the `minismlsharp` LLVM IR file, link them with the runtime library, and create a `minismlsharp` command.
4. By using this `minismlsharp` command, compile all of the tools and libraries, and the full-featured SML# compiler. This procedure is roughly performed as follows.
   (a) Compile the Basis Library.
   (b) Compile and link `smllex` command.
   (c) Compile and link ML-yacc library and `smlyacc` command.
   (d) Generate parser source code by `smllex` and `smlyacc` command.
   (e) Compile and link `smlformat` command.
   (f) Generate printer source code by `smlformat` command.
   (g) Compile all of the libraries bundled to the SML# distribution.
   (h) Compile and link `smlsharp` command.
5. The following files are installed to specified destination directories.
   - The static link library of the runtime library.
   - Interface files, object files, and signature files of the libraries bundled to the SML# distribution.
   - `smllex`, `smlyacc`, `smlformat`, `smlsharp` command.

As outlined above, there are complex dependencies among source files and commands. Furthermore, processing some of these files depend on the underlying OS. This is a typical situation in a large system development. One well established method to solve these dependency problems is to use `configure` script generated by GNU Autoconf and `make` command.

SML# compiler compiles each source file according to its interface file, which describes the set of files require by the source file. SML# compiler can also generate a list of files on which each source file depends in the `Makefile` format that can be processed by `make` command. SML# compiler does this task, when it is given one of the following switch.

1. `smlsharp -MM` *smlFile*. The compiler generates the dependency for the source file *smlFile* to be compiled in the Makefile format.

2. `smlsharp -MMl` *smiFile*. The compiler assumes that the file *smiFile* specifies the top level system, and generates the list of necessary object files in the Makefile format.

3. `smlsharp -MMm` *smiFile*. The compiler assumes that the file *smiFile* specifies the top level system, and generates a Makefile that builds the executable file of the entire program.

In the SML# project, we make a `Makefile` that performs the above described complicated sequence of compilation and linking steps using the above functionality of SML#. Invoking `make` command on `Makefile` re-compiles only the necessary files to build SML# compiler.

6.3 Let's try SML# interactive mode

Include the SML# installation directory in your command load path, so that you can run SML# by the name `smlsharp`. When invoked without any parameter, SML# stars its interactive session by printing the following message and waits for your input.

```
$ smlsharp
SML# version 4.2.0 (2025-03-24) for x86_64-pc-linux-gnu with LLVM 20.1
#
```

The "`# `" character is the prompt SML# prints. In this document, we write "`$ `" for the shell prompt.

After this message, the compiler repeats the following steps.

1. Read the user input up to "`;`".

2. Compile the input program and execute it.

3. Print the result.

The following is a simple example.

```
# "Hello world";
val it = "Hello world" : string
```

The first line is the user input. The second line is the response of SML# system. As seen in this example, the compiler prints the result value with its type, and binds it to a name for subsequent use. If the user does not specify the name, the compiler uses "`it`" as the default name.

6.4 Let's try SML# compile mode

The main function of SML# compiler is to compile a file and to create an executable program, just like `gcc`.

As a simple example, let us write the previous user input `# "Hello world";` into a file `hello1.sml`:

```
"Hello world";
```

The trailing semicolon does not have any effect in a file. This file can be compiled as follows.

```
$ smlsharp hello1.sml
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=43a2c24d3728ad6a35262f
not stripped
$
```

When only a file name is given, SML# compiles the file, link it and creates an executable file, whose default name is `a.out`. As shown by the Linux `file` command, it is an ordinary executable file. The output file name can be specified by a `-o` switch.

```
$ smlsharp -o hello1 hello1.sml
$ ls hello1*
hello1 hello1.sml
$
```

The generated executable file can then be executed.

```
$ ./hello1
$
```

Nothing is printed by this program. This is what you expect. SML# compiles the source code itself; it does not attach code to print the result value and its type. If you want to see something printed, you need to write code to print it explicitly. Now let's create a file `hello2.sml` to print this message. The contents can be the following.

```
print "hello world!\n"
```

This file contains a function `print` which is not defined in this file. Our intention is that `print` denotes the function `print : string -> unit` in Standard ML Basis Library. However, since name `print` can be freely re-defined, for SML# compiler, it does not necessarily mean the library function. In order to compile `hello2.sml`, you need to notify the compiler where the name `print` is defined. For this purpose, an **interface file** must be created.

When compiling `hello2.sml`, SML# searches for its interface file by its default name `hello2.smi`. So to compile `hello2.sml`, you need to create file `hello2.smi`. Its contents can be the following.

```
_require "basis.smi"
```

This declares that `hello2.sml` uses names that are defined in the interface file `"basis.smi"`, which is the system supplied file containing the declarations of all the names defined in the Standard ML Basis Library.

With this preparation, `hello2.sml` is compiled as follows.

```
$ smlsharp hello.sml -o hello
$ ./hello
hello world!
```

6.5 `smlsharp` command modes

`smlsharp` has the following execution modes.

interactive mode When `smlsharp` is invoked without any parameter, it executes the interactive session.

compile and link mode When `smlsharp` is invoked with a single source file as shown below:

```
$ smlsharp ⟨file⟩.sml
$
```

it compiles the source file ⟨*file*⟩`.sml`, links the resulting object file and creates an executable program (with the name `a.out` by default).

compile mode When `smlsharp` is invoked with `-c` switch as shown below:

```
$ smlsharp -c ⟨file1⟩.sml ··· ⟨fileN⟩.sml
$
```

it compiles the given files ⟨*file1*⟩`.sml` ··· ⟨*fileN*⟩`.sml` into object files ⟨*file1*⟩`.o` ··· ⟨*fileN*⟩`.o`.

link mode When `smlsharp` is invoked with a single interface files as shown below:

```
$ smlsharp ⟨file⟩.smi
$
```

it assumes that ⟨*file*⟩`.smi` is a top-level interface file, links all the object files corresponding to the interface files referenced from ⟨*file*⟩`.smi`, and generates an executable file.

SML# also recognizes various command-line switches.

```
$ smlsharp --help
```

will print a help message describing available switches.

Chapter 7

Introduction to ML programming

The SML# language is a super set of Standard ML. To write a program in SML# exploiting its advanced features, you must first learn Standard ML programming. This chapter provides ML programming tutorial for those who first learn ML programming.

7.1 About the ML language family

SML# is a programming language in the ML family.

ML was first developed as a **M**meta **L**anguage of Edinburgh LCF[3] system. The prefix *meta* in this particular usage probably traces back to Greek phrase "ta meta ta phusika", which simply denotes the book placed after the phusika (Physics, or the book on the nature). The denoted book happens to be on philosophy (metaphysics), which made this prefix receive a meaning of the (philosophical) attitude of analyzing things beyond the ordinary way of thinking. In the context of languages, a meta language is a language used to analyze the ordinary usage (writing/reading/speaking) of a language. Objects of LCF system are computable functions (programs) represented by a language called PPLAMBDA. LCF ML was the meta language of PPLAMBDA, namely, a programming language to manipulate PPLAMBDA expressions (functions).

ML's name as well as its features are originated from this historical role of LCF ML. For flexible manipulation of programs, LCF ML itself was defined as a functional language. In addition, a type inference system was introduced for reliable manipulation of function terms. Since the success of LCF ML, ML has evolved as a general purpose programming language, and several compilers have been developed for ML. Based on these efforts, its specification is formally defined as Standard ML [5, 6].

7.2 Declarative programming

Functional programming is sometimes described as "declarative programming". This is not a precise technical term. It somewhat vaguely suggests that programs naturally and directly express what they should realize. If the problem to be solved is procedural in nature then a procedural description may well be declarative. However, when the thing to be represented by a program has a well defined meaning, then declarative programming has more precise meaning.

When a program to be written is best understood as a function that takes an input and returns a result, then functional programming would be naturally declarative. As a simple example, consider the factorial function. The factorial of n, $n!$, is defined by the following mathematical equations.

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \end{aligned}$$

In ML, it is coded below.

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)
```

This declaration defines a function named `fact`. It consists of two cases separated by "`|`". The first case says `fact` returns `1` if the parameter is `0`, and the second case says it otherwise returns the multiple of

`n` and the factorial of `n - 1`. These two cases exactly correspond to the mathematical definition of this function.

Various programs other than such a very simple mathematical function can be written in a concise and readable fashion if you can represent them as declarative functions according to the intended meaning. ML is a programming language that promotes this way of declarative programming. A key to master ML programming is to obtain a skill of writing declarative code in this sense. In the subsequent sections, we learn the essence of ML programming.

7.3 Representing computation by composing expressions

In the factorial example, the program returns `1` if the parameter is `0`, and returns `n * fact (n - 1)` otherwise. In this way, the value returned by a function is represented by an *expression*. `1` in the first case is an expression representing the natural number 1. `n * fact (n - 1)` is an expression involving variable `n` and function application. The basic principle of ML programming is

programming is done by defining an expression that represents the desired value.

An expression consist of the following components:

1. constants such as `1`,
2. variables representing function parameters and defined values.
3. function applications (function calls), and
4. functions and other data structure constructions

The items 1 through 3 are the same as in mathematical expressions, we have leaned in school. For example, the sum of the arithmetic sequence $S_n = 1^2 + 2^2 + \cdots + n^2$ is given by the following equation.

$$S_n = \frac{n(n+1)(2n+1)}{6}$$

This is directly programmed as follows.

```
val Sn = (n * (n + 1) * (2 * n + 1)) div 6
```

`*` and `div` are integer multiplication and division, respectively. When n is defined, it correctly computes S_n as seen below.

```
# val n = 10;
val n = 10 : int
# val Sn = (n * (n + 1) * (2 * n + 1)) div 6;
val Sn = 385 : int
```

7.4 Constants literals and built-in primitives

SML# contains the following atomic types, with the associated literal constants.

type	its values
`int`	signed integers (of one machine word size)
`real`	double precision floating point numbers
`char`	characters
`string`	character strings
`word`	unsigned integers (of one machine word size)
bool	boolean values

We review basis expressions for these atomic types, except for `bool` which will be treated in the next section.

7.4.1 int type

This is a standard type of integers. In SML#, a value of `int` is a two's compliment integer and is the same as `long` in C. Constants of `int` are written either the usual decimal notation or hexadecimal notation of the form `0x`$\langle h \rangle$ where $\langle h \rangle$ is a sequence of digits from `0` to `9` and letters from `a`(or `A`) to `f`(or `F`). A negative constant is written with `~`. SML# prints `int` values in decimal notation.

```
# 10;
val it = 10 : int
# 0xA;
val it = 10 : int
# ~0xFF;
val it = ~255 : int
```

Binary arithmetic operations `+, -, *, div` are defined on `int`.

```
# 10 + 0xA;
val it = 20 : int
# 0 - 0xA;
val it = ~10 : int
# 10 div 2;
val it = 5 : int
# 10 * 2;
val it = 20 : int
```

As in arithmetic expressions in mathematics, `*,div` associate stronger than `+, -`, and operators of the same strength are computed from left to right.

```
# 1 + 2 * 3;
val it = 7 : int
# 4 - 3 - 2;
val it = ~1 : int
```

7.4.2 real type

`real` is a type of floating point data. In SML#, a value of `real` is a 64 bit double precision floating point numbers and is the same as `double` in C. `real` constants are written in the notation $\langle s \rangle$ `.` $\langle d \rangle$ `E` $\langle s \rangle$, where $\langle s \rangle$ is a decimal string possibly with the negation symbol and $\langle d \rangle$ is a decimal string. Each of the three parts can be omitted, but decimal string is interpreted as a value of `int`.

```
# 10.0;
val it = 10.0 : real
# 1E2;
val it = 100.0 : real
# 0.001;
val it = 0.001 : real
# ~1E~2;
val it = ~0.01 : real
# 10;
val it = 10 : int
```

Binary arithmetic operators `+, -, *, /` are defined for `real`. Note that the division operator on `real` is `/` and not `div`.

```
# 10.0 + 1E1;
val it = 20.0 : real
# 0.0 - 10.0;
val it = ~10.0 : real
# 10.0 / 2.0;
val it = 5.0 : real
# 10.0 * 2.0;
val it = 20.0 : real
```

7.4.3 char type

`char` is a type for one character data. A constant for a character $\langle c \rangle$ is written as `#"` $\langle c \rangle$ `"`. $\langle c \rangle$ may be any printable character or one of the following special escape sequences.

`\a`	warning (ASCII 7)
`\b`	backspace (ASCII 8)
`\t`	horizontal tab(ASCII 9)
`\n`	new line (ASCII 10)
`\v`	vertical tab (ASCII 11)
`\f`	home feed (ASCII 12)
`\r`	carriage return(ASCII 13)
`\^`C	control character C
`\"`	character `"`
`\\`	character `\`
`\`*ddd*	the character whose code is *ddd* in decimal

```
# #"a";
val it = #"a" : char
# #"\n";
val it = #"\n" : char
```

The following primitive functions are defined on `char`.

```
val chr : int -> char
val ord : char -> int
```

`chr` n assumes that n is an ASCII code and returns the character of that code. `ord` c returns the ASCII code of c.

```
# ord #"a";
val it = 97 : int
# chr(97 + 7);
val it = #"h" : char
# ord #"a" - ord #"A";
val it = 32 : int
# chr(ord #"H" + 32);
val it = #"h" : char
```

7.4.4 string type

`string` is a type for character strings. `string` constants are written with `"` and `"`. A `string` constant may contain escape sequences shown above. For `string` data, a print function `print` and a concatenation binary operator `^` are defined.

```
# "SML#" ^ "\n";
val it = "SML#\n" : string
# print it;
SML#
val it = () : unit
```

7.4.5 word type

`word` is a type for unsigned integers. `word` constants are written as `0w` $\langle d \rangle$ or `0wx` $\langle h \rangle$ where $\langle d \rangle$ is a decimal digit sequence and $\langle h \rangle$ is a hexadecimal digit sequence. SML# prints `word` data in a hexadecimal notation.

```
# 0w10;
val it = 0wxa : word
# 0wxA;
val it = 0wxa : word
```

7.5 Type bool and conditional expressions

Following the ML principle,

programming is done by defining an expression that represents the desired value.

conditional computation is represented as an expression. A conditional expression

`if` E_1 `then` E_2 `else` E_3

is an expression that computes a value in the following steps.

1. Evaluate $E1$ and obtain a value.
2. If the value is `true` then evaluate $E2$ and return its value.
3. If the value of $E1$ is `false` then evaluate $E3$ return its value.

`true`, `false` are two constants of type `bool`. Expressions of type `bool` contains comparison expressions and logical operations.

```
# 1 < 2;
val it = true : bool
# 1 < 2 andalso 1 > 2;
val it = false : bool
# 1 < 2 orelse 1 > 2;
val it = true : bool
```

Conditional expressions are made up with expressions of type `bool`.

```
# if 1 < 2 then 1 else 2;
val it = 1 : int
```

Since this is an expression, it can be combined with other expressions as show below.

```
# (if 1 < 2 then 1 else 2) * 10;
val it = 10 : int
```

7.6 Compound expressions and function definitions

Of course, constant, variables and primitive functions are not sufficient for writing a program that solves a complex problem. The important expressions in ML programming are those that represent functions.

As seen in an example in Section 7.2, a function is defined by the following syntax:

fun *funName param* `=` *expr*

After this declaration, the variable *funName* is bound to a function that takes *param* as its argument and returns the result computed by *expr*. For example, the equation for an arithmetic sequence we have seen before can be regarded as a function that takes a natural number n:

$$S(n) = \frac{n(n+1)(2n+1)}{6}$$

This definition can be programmed directly using `fun` as follows.

```
# fun S n = (n * (n + 1) * (2 * n + 1)) div 6;
val S = fn : int -> int
```

After this definition, `S` can be used as follows.

```
# S 10;
val it = 385 : int
# S 20;
val it = 2870 : int
```

7.7 Recursive functions

The `fun` syntax allows recursive function definitions. The function name *funName* being defined in this declaration can be used in the defining body *expr* of this declaration.

A recursive function can naturally be designed in the following step.

1. Presume the expected behavior of the function *funName* for all the cases, and assume that such function exists.
2. For each case of the parameter `param`, write the *expr* using `param` and `funName` for that case.

For example, function `fact` shown in Section 7.2 is designed in the following steps.

1. Assume that you have the function `fact` that correctly computes the factorial of any non-negative number.
2. Using `fact`, write down the body of each case as follows.
   (a) If the parameter is `0`, then since $0! = 1$, the `expr` is `1`,
   (b) otherwise, from the definition of the factorial function, the `expr` is `n * fact(n - 1)`.

This yields the following recursive function definition.

```
# fun fact 0 = 1
>   | fact n = n * fact (n - 1);
val fact = fn : int -> int
```

The summation function `sum` and exponentiation function `power`, for example, can similarly be written by presuming that these functions were already given.

```
fun sum 0 = 0
  | sum n = n + sum (n - 1)
fun power 0 = 1
  | power n = c * power (n - 1)
```

Under the presumed behavior of `sum` and `power`, each case correctly computes the expected value, and therefore the entire definitions are correct.

This is a recursive way of reasoning. Think in this recursive way, and you can naturally write various recursive functions.

7.8 Functions with multiple arguments

The function `power` defined in the previous section computers the power C^n of C for a given `C`. There are two ways of generalizing this function so that it takes both `n` and `C` and return C^n. They are shown in the following interactive session.

```
# fun powerUncurry (0, C) = 1
>   | powerUncurry (n, C) = C * powerUncurry (n - 1, C);
val powerUncurry = fn : int * int -> int
# powerUncurry (2,3);
val it = 9 : int

# fun powerCurry 0 C = 1
>   | powerCurry n C = C * (powerCurry (n - 1) C);
val powerCurry = fn : int -> int -> int
# powerCurry 2 3;
val it = 9 : int
```

The type `int * int -> int` of `powerUncurry` indicates that it is a function that takes a pair of integers and returns an integer. In contrast, the type `int -> int -> int` of `powerCurry` says that it is a function that takes an integer and returns a function of type `int -> int`. As seen in these example, in ML, a function is defined in the format as it will be used. For example, `powerUncurry` is defined `fun powerUncurry (C,n) = ...` so it is used as `powerUncurry (2, 3)`.

7.9 Function application syntax

`powerUncurry` is just like a C function taking a pair of arguments. Unlike C and other procedural languages, ML also allows the programmer to write a function that takes multiple arguments in a sequence like `powerCurry`. For proper understanding of this feature, let us review function application syntax. In mathematics, function application is written as $f(x)$, but in ML it is written as `f x`, simply juxtaposing function and its argument.

Let *expr* be the expressions consisting of constants, variables and function applications. Its syntax is given by the following grammar.

$$\begin{array}{lcll} expr & ::= & c & \text{(constants)} \\ & | & x & \text{(variables)} \\ & | & expr\ expr & \text{(function application)} \\ & | & \cdots & \end{array}$$

This grammar is ambiguous in the sequence of function applications. To understand this problem, let us examine familiar arithmetic expressions. Suppose we introduce $expr + expr$, $expr - expr$, and $expr * expr$ for integer addition, subtraction and multiplication. Then we can write `10 - 6 - 2` but this expression itself does not determine the order of subtractions. As a consequence, two different results are possible: `(10 - 6) - 2 = 2` or `10 - (6 - 2) = 4`. As in elementary school arithmetic, ML interprets this as `(10 - 6) - 2 = 2`. We say that the subtraction *associates to the left* or *is left associative*. For anther example, ML interprets `10 + 6 * 2` as `10 + (6 * 2)`. We say that the multiplication `*` associates *stronger than* addition `+`.

In ML and other lambda calculus based functional languages, there is the following important rule:

> function application associates to the left and its associatibity is the strongest

`e1 e2 e3` $\cdots$ `en` is interpreted as `(`$\cdots$ `((e1 e2) e3)` $\cdots$ `en)`. For example, `powerCurry 2 3` is interpreted as `((powerCurry 2) 3)`.

7.10 Higher-order functions

As we learned that `powerCurry 2 3` is interpreted as `((powerCurry 2) 3)`. This implies:

> `(powerCurry 2)` is a function.

In ML, functions can be generated by a program. `powerCurry` is a program that takes an integer and return a function. Generated functions can be named and passed to other functions. For example, one can write the following code.

```
# val square = powerCurry 2;
val square = fn : int -> int
# square 3;
val it = 9 : int
# fun apply f = f 3;
val apply = fn : ['a. (int -> 'a) -> 'a]
# apply square;
val it = 9 : int
```

The type of `apply` means that this is a function that takes a function on integers and return an integer.

In summary, ML has the following power.

> Just like integers, functions are values that can be returned from a function or passed to a function. In particular, a function can take a function as a parameter.

This is the important principle of ML under the fundamental principle of ML programming saying that programs are expressions. A function that takes a function as a parameter or return a function is called a **higher-order function**.

7.11 Using higher-order functions

The key to writing a cool, i.e. readable and concise program efficiently is to understand the role of higher-order functions and to compose expressions using higher-order functions. As in any programming skill, mastering higher-order functions require certain amount of practice, and is beyond this tutorial. However, there are few basic concepts that are important in writing expressions using higher-order functions. This tutorial attempt to exhibit them. In what follows, we learn these concepts through simple examples.

In high school mathematics, we have learned Σ notation. This notation satisfies the following equations.

$$\begin{aligned}\sum_{k=1}^{0} f(k) &= 0 \\ \sum_{k=1}^{n+1} f(k) &= f(n) + \sum_{k=1}^{n} f(k)\end{aligned}$$

The reason for learning this notation is because this notation represents a useful abstraction, i.e. summing up a sequence of expressions with integer indexes. A brief analysis of the notation $\sum_{k=1}^{n} f(k)$ reveals the following.

- The notation contains 3 variables k, n, f. Among them, k is a variable that is only to indicate that expressions are index by $k = 1, \ldots, n$, and is not a real variable. The variables of this expression are n and f.
- Expression $\sum_{k=1}^{n} f(k)$ denotes the value $f(1) + f(2) + \cdots + f(n)$ for any given integer n and a function f.

$\sum$ is a function that takes an integer n and a function f that takes an integer and returns an integer, i.e. it is a higher-order function. In ML, this higher-order function is directly code as follows.

```
# fun Sigma f 0 = 0
>   | Sigma f n = f n + Sigma f (n - 1);
val Sigma = fn : (int -> int) -> int -> int
```

This `Sigma` can compute the summation of any inter function.

```
# Sigma square 3;
val it = 14 : int
# Sigma (powerCurry 3) 3;
val it = 36 : int
```

As demonstrated by this example, higher-order functions has the power of representing a useful computation pattern by abstracting its components as argument functions. The key to writing a cool ML code is to master the skill of abstracting useful computation patterns as higher-order functions. Polymorphic typing we shall learn in Section 7.20 can be regarded as a mechanism to support this style of programming. Higher-functions with their polymorphic typing enable the ML programmer to code a complicated system in a concise and declarative way.

7.12 Imperative features of ML

ML is a language where programs are defined by composing expressions, and is suitable for declarative programming. So far we have used numerical functions such as $\sum$, but as pointed out in Section 7.2, the intention of declarative programming is to write what the program should express directly as a code, and not intend to define a program as a mathematical function.

If the actual problem is best expressed as a sequence of procedure, then directly writing done that sequence would be most declarative. For example, displaying a sentence on the video display you are looking at is the process to override the previous sentence with the new sequence of characters. In an actual implementation, this is done by modifying the frame buffer of the video screen hardware. This process is inherently procedural, and therefore best described as a process to update the buffer.

Input from and output to external world inevitably have this property. For those problems, procedural representation would be most direct and therefore concise and readable, thus declarative.

Integrating imperative features in a functional language requires the following.

1. Introduction of updating a mutable state. In the previous display example, display contents can be understood as the state of the display. The basis of imperative programming is to update states to obtain the desired state. For this, the introduction of mutable data structures is required.

2. A predictable evaluation order. In the display example, the order of updates determine the image and its motion in the display. To write a procedural program, it is necessary to control the order of update operations.

ML introduces these two in a framework of functional programming.

7.13 Mutable memory reference types

ML has the following type constructor and primitives.

```
type 'a ref
val ref : 'a -> 'a ref
val ! : 'a ref -> 'a
val := : 'a ref * 'a -> unit
```

`'a ref` is a type of references (pointers) of type `'a`. Function `ref` takes a value of type `'a` and returns a reference to it. Function `!` takes a reference and returns the value of that reference. Function `:=` assigns a given value to a given reference. This destructively update the reference cell. The following shows an interactive session manipulating a reference.

```
# val x = ref 1;
val x = ref 1 : int ref
# !x;
val it = 1 : int
# x := 2;
val it = () : unit
# !x;
val it = 2 : int
```

A function performing imperative operation can be written by maintaining a state using a reference.

7.14 Left-to-right applicative order evaluation

To write a program that manipulate states, it is necessary to fix the order of execution of each part of the program. In a functional language, where a program is a composition of expression, program execution corresponds to obtain the value of an expression. This process is called *evaluation of an expression*. In ML, expressions are evaluated in the following order.

1. The same level sub-expressions are evaluated from left to right. For example, `(power 2) (power 2 2)` is evaluated by first evaluating `(power 2)` to obtain a function that computes power of 2 and then `(power 2 2)` is evaluated to obtain 4, and finally the power 2 function is applied to 4 to yield 14.

2. A sequence of declarations are evaluated in the order of the declarations.

3. The body of a function is not evaluated until the function is applied to an argument.

ML also provide the following syntax to control evaluation order.

$$
\begin{array}{lcll}
\mathit{expr} & ::= & \cdots & \\
 & | & \texttt{(}\mathit{expr}\texttt{; } \cdots \texttt{ ;}\mathit{expr}\texttt{)} & \text{(sequential evaluation)}
\end{array}
$$

(*expr*$_1$; $\cdots$; *expr*$_n$) is evaluated by evaluating *expr*$_1$ through *expr*$_n$ sequentially, and yields the value of the last expression *expr*$_n$.

By manipulating reference types in the evaluation order, an imperative operation can be written as a function. For example, a function that generate a new name by maintaining a mutable state is defined as follows.

```
fun makeNewId () =
  let
    val cell = ref 0
    fun newId () =
      let
        val id = !cell
      in
        (cell := id + 1; id)
      end
  in
    newId
  end
```

7.15 Procedural control

We have learned imperative (procedural) programming. At this point, let us review the relation between control statements in a procedural language and functions.

Inherently procedural operations such as external IO are naturally expressed as procedural programming. However, in a procedural language, control structures such as loops and conditionals are also implemented using states. Since program structures are independent of imperative operation, representing them as expressions generally yield more declarative programs.

There is one thing to be noted however. In a procedural language like C, the factorial function may be written as follows.

```
int fact (int n) {
  int s = 1;
  while (n > 0) {
    s = s * n;
    n = n - 1;
  }
  return s;
}
```

This is generally more efficient than a recursive function of the form

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)
```

This fact is however does not implies that functional programming is inherently inefficient.

7.16 Loop and tail recursion

The basis of designing a loop program are the following.

1. Design a state of computation that leads to the desired result.
2. Set up variables that hold the computation state, and the progress of computation.
3. Write a code to iteratively update the variables until the desired result is obtained.

A program that sums up 1 through n can be designed as follows.

- the state of computation: $s = i * (i + 1) * ... * n (1 \leq i \leq n)$
- variables : s, i

- update code: $s := s + 1; i := i - 1$

This design yields the C function `fact` in Section 7.15.

This way of designing an iterative computation is not limited to procedural languages. A code to update the current state can be understood as a function that takes the current state and generate the next state. Then a loop program is represented a recursive function that takes the current state and call itself with the generated next state. The loop code of the C function `fact` can be written as the following ML code:

```
fun loop (0, s) = s
  | loop (n, s) = loop (n - 1, s * n)
fun fact n = loop (n, 1)
```

This form of recursion is called `tail recursion`, which is evaluated efficiently.

7.17 let expressions

ML programming is done by defining an expression and giving a name to it for subsequent use. Examples we have seen so far, names are all recorded at the top-level. However, a large program requires a large number of names, many of which are only temporarily used. For example, `fact` function in the previous section is defined using `loop` function, but this name `loop` is only used in `fact`; other function would need another version of `loop`. Such a name should be defined as a local name to the function that use the name. For this purpose, ML provide the following let expression.

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \cdots \\
 & | & \texttt{let}\ \mathit{decl\_list}\ \texttt{in}\ \mathit{expr}\ \texttt{end} \\
\mathit{decl} & ::= & \texttt{val}\ x\ \texttt{=}\ \mathit{expr} \\
 & | & \texttt{fun}\ f\ p_1\ \cdots\ x_n\ \texttt{=}\ \mathit{expr} \\
 & | & \cdots
\end{array}
$$

The declarations written between `let` and `in` are only visible to the code between `in` and `end`. A tail recursive `fact` function is usually written as follows.

```
fun factorial n =
  let
    fun loop (s, 0) = s
      | loop (s, i) = loop (s * i, i - 1)
  in
    loop (1, n)
  end
```

7.18 List data type

Lists and the associated functions are commonly used data structures in ML. List processing programs contain the following basic elements in ML programming.

- Recursively defined data.
- Pattern matching.
- Polymorphic functions

Let us lean these elements through list programming.

A list is a sequence of elements. In ML, a list of 1,2,3 is written as follows.

```
# [1,2,3];
val it = [1, 2, 3] : int list
```

This notation is a shorthand for the following expression.

```
# 1 :: 2 :: 3 :: nil;
val it = [1, 2, 3] : int list
```

`::` is a right-associative binary operator for constructing a list. So `1::2::3::nil` is interpreted as `1::(2::(3::nil))`. e `::` L is the list obtained by pre-pending the element `e` to the list `L`. `nil` is the empty list. `int list` is the list type whose component is type `int`.

7.19 Principle in composing expressions

The fundamental principle of ML programming to compose expressions, but of course not all compositions yield meaningful programs. In ML, the principle of composing expressions is the following.

expressions are freely composed as far as they are type correct.

Let us consider this principle using lists as examples. List does not constrain its component types; any values can be put into a list as far as their type is the same. The following interactive session shows constructions of lists of various types.

```
  # fact 4 :: 4 + 4 :: (if factorial 1 = 0 then nil else [1,2,3]);
  val it = [12, 24, 8] : int list
  # [1.1, Math.pi, Math.sqrt 2.0];
  val it = [1.1, 3.14159265359, 1.41421356237] : real list
  # "I"::"became"::"fully"::"operational"::"on"::"April 2, 2012"::nil;
  val it = ["I", "became", "fully", "operational", "on", "April 2, 2012"] : string
list
  # [factorial, fib];
  val it = [fn, fn] : (int -> int) list
  # [#"S", #"M", #"L", #"#"];
  val it = [#"S", #"M", #"L", #"#"] : char list
  # implode it;
  val it = "SML#" : string
  # explode it;
  val it = [#"S", #"M", #"L", #"#"] : char list
```

"implode" convert list of characters into a string and "explode" does its converse.

7.20 Polymorphic functions

In order to exploit the principle that "expressions are freely composed as far as they are type consistent", primitive functions used to compose expressions should accept expressions of various types. The primitive `::` to construct a list has the following type.

```
# op ::;
val it = fn : ['a. 'a * 'a list -> 'a list]
```

`op` prefix convert infix binary operator into a function that takes a pair. This typing indicates that `::` is a function that takes a value of type `'a` and a value of type `'a list`, which is a list type whose element type is `'a`, and returns a list of the same type. In this typing, `'a` represents an arbitrary type. The notation `['a.`$\cdots$`]` indicates that `'a` in $\cdots$ can be replaced with any type, and corresponds to universally quantified formula $\forall a.\cdots$ in logic. These types that quantifies over type variables are called *polymorphic types*, and functions having a polymorphic type are called **polymorphic functions**.

Functions defined by composing polymorphic functions are often polymorphic functions.

```
# fun cons e L = e :: L;
val cons = fn : ['a. 'a -> 'a list -> 'a list]
```

In this way, ML compiler infers **a most general polymorphic type** for an expression. To understand this mechanism, consider the following function.

```
fun twice f x = f (f x);
```

`twice` takes a function and an argument and apply the function to the argument twice. Since `f` is applied to `x`, the type of `x` must be the same as the argument type of `f`. Moreover, since `f` is applied again to the result of `f`, the result type of `f` must be the same as its argument type. The most general type satisfying these constraint is the following.

```
['a. ('a -> 'a) -> 'a -> 'a]
```

ML indeed infers the following typing for `twice`.

```
# fun twice f x = f (f x);
val twice = fn : ['a. ('a -> 'a) -> 'a -> 'a]
```

ML's principle in program construction – "expressions are composed as far as they are type consistent" – is made possible by this polymorphic type inference mechanism. `twice` can be combined with any function and a value as far as the function return the same type as its argument and that the value has the argument type of the function. This freedom and constraint is automatically guaranteed by the inferred typing of `twice`.

Chapter 8

SML# feature: record polymorphism

The rest of this part introduce the extensions newly introduce by SML# through examples.

In this chapter, we learn programming with records based on record polymorphism.

We note that record polymorphism is not a special additional feature in record programming, but the basic mechanism required to realize the ML's principle in program construction: "expressions are composed as far as they are type consistent". Standard ML lacks this basic mechanism and therefore in Standard ML, one cannot write ML-style program in manipulating records. For this reason, we postpone explanation of records until this chapter.

Let's start learning record programming basics.

8.1 Record expressions

The syntax of record expressions is given below.

$$\begin{array}{lll} expr & ::= & \cdots \\ & | & \{l_1\texttt{=}expr_1\texttt{,}\cdots\texttt{,}\ l_n\texttt{=}expr_n\} \end{array}$$

l denotes a string called *labels*. Below is a simple record expression.

```
# val point = {X =0.0, Y=0.0};
val point = {X = 0.0, Y = 0.0} : {X: real, Y: real}
```

A record whose labels are consecutive numbers starting with `1` is interpreted as a tuple and printed specially.

```
# {1 = 1.1, 2 = fn x => x + 1, 3 = "SML#"};
val it = (1.1,fn,"SML#") : real * (int -> int) * string
# (1,2);
val it = (1,2) : int * int
```

In Section 7.8, we defined a multiple-argument function, namely `powerUncurry (n,C)`. This is a function that takes a tuple.

Just like lists, record elements can contain values of any types. A record forming function is therefore polymorphic, as seen in the following example.

```
# fun f x y = {X = x, Y = y};
val f = fn : ['a. 'a -> ['b. 'b -> {X: 'a, Y: 'b}]]
# fun g x y = (x, y);
val g = fn : ['a. 'a -> ['b. 'b -> 'a * 'b]]
```

8.2 Field selection operation

The basic operation on records is to select a field value by specifying a field label. For this, the following primitive function is defined for any label l.

`#`*l*

This function takes a record containing a field labeled with *l*, and returns the value of that label. According to the ML principle, this function should be applicable to any record containing label *l*. For this primitive, SML# infers the following polymorphic typing.

```
# #X;
val it = fn : ['a#{X: 'b}, 'b. 'a -> 'b]
```

The notation `'a#{X:'b}` is a type variable `'a` which represents an arbitrary record type that contains a field labeled `X` of type `'b`. The inferred type is a most general type of `#X`. Below show some examples.

```
# #X {X=1.1, Y=2.2};
val it = 1.1 : real
# #X {X = 1, Y = 2, Z = 3};
val it = 1 : int
```

Functions using these operations are polymorphic in record structures, as seen in the following example.

```
# fun f x = (#X x, #Y x);
val f = fn : ['a#{X: 'b, Y: 'c}, 'b, 'c. 'a -> 'b * 'c]
```

The type of `f` indicates that this is a function that takes any record containing `X:'b` and `Y:'c` fields and returns a pair of type `'b * 'c`. This is a most general polymorphic type of this function. So this function can be freely combined with other expressions as far the combination is type consistent.

8.3 Record patterns

Record field selection can also be done through pattern matching. Standard ML has the following patterns.

$$
\begin{array}{rcl}
pat & ::= & \cdots \\
 & | & \{field\_list\} \\
 & | & \{field\_list, \ldots\} \\
field & ::= & l\texttt{=}pat \mid l
\end{array}
$$

The first pattern matches a record having the specified set of labels, and the second pattern matches any record containing the set of specified labels. When only a label is specified in a field then it is interpreted as a variable with the same name is specified. For example, a pattern `{X, Y}` is interpreted as `{X = X, Y = Y}`. The following are examples of field selection through record patterns.

```
# fun f {X = x, Y = y} = (x, y);
val f = fn : ['a, 'b. {X: 'a, Y: 'b} -> 'a * 'b]
# fun f {X = x, Y = y, ...} = (x, y);
val f = fn : ['a#{X: 'b, Y: 'c}, 'b, 'c. 'a -> 'b * 'c]
# fun f {X, Y, ...} = (X, Y);
val f = fn : ['a#{X: 'b, Y: 'c}, 'b, 'c. 'a -> 'b * 'c]
```

Record pattern can be freely combined with other patterns.

```
# fun f ({X,...}::_) = X;
val f = fn : ['a#{X: 'b}, 'b. 'a list -> 'b]
```

In this example, `f` takes a list of records containing `X` field, and returns the `X` field of the first record in the list.

8.4 Functional record update

SML# contains functional record update expressions, whose syntax is given below.

$$\begin{array}{lcl} expr & ::= & \cdots \\ & | & expr \texttt{ \# \{}l_1\texttt{=}expr_1\texttt{,}\cdots\texttt{,}\ l_n\texttt{=}expr_n\texttt{\}} \end{array}$$

This expression creates a new record by modifying the value of each label l_i to $expr_i$. This is an expression to create a new record; the original record *expr* is not mutated. This expression has a polymorphic type according to the ML's principle of most general typing. Below is an example using this expression.

```
# fun f modify x = modify # {X = x};
val f = fn : ['a#{X: 'b}, 'b. 'a -> 'b -> 'a]
```

The following is a useful idiom in record programming.

```
# fun reStructure (p as {Salary,...}) = p # {Salary = Salary * (1.0 - 0.0803)};
val reStructure = fn : ['a#{Salary: real}. 'a -> 'a]
```

Function `reStructure` takes an employee record `p` and reduces its `Salary` field by 8.03%. As seen in its typing, this function can be applied to any record as far as it contains a , `Salary` field.

8.5 Record programming examples

In a language supporting record polymorphic (currently SML# seems to be the only one), one can write generic code by focusing only on relevant properties of problems. This provide a powerful tool for modular construction of programs that scales to large software development.

To understand a flavor of this feature, let us consider a problem to simulate object movement in a parabolic path in a Cartesian coordinate system. An object can be represented as a record containing `X:real` and `Y:real` fields representing the current position vector, and `Vx:real` and `Vy:real` fields representing the current velocity vector. An object may have a lot of other properties, but for writing a program to simulate objects movement, these attributes are sufficient. Object movement is simulated by repeatedly applying a function `move` that moves an object from the current location to the location after one unit of time.

```
val move : ['a#{X:real, Y:real, Vx:real, Vy:real}. 'a * real -> 'a]
```

In this exercise, we assume that unit of time is 1 second. In the Cartesian coordinate system, since a position vector and a velocity vector are compositions of those of the two coordinates, we can write write functions on `X` and `Y` coordinates independently and compose them. The next position is obtained by adding the velocity. So we can write functions on `X` and `Y` independently as below (which shows the code and its typing in an interactive session).

```
# fun moveX (p as {X:real, Vx:real,...}, t:real) = p # {X = X + Vx};
val moveX = fn : ['a#{Vx: real, X: real}. 'a * real -> 'a]
# fun moveY (p as {Y:real, Vy:real,...}, t:real) = p # {Y = Y + Vy};
val moveY = fn : ['a#{Vy: real, Y: real}. 'a * real -> 'a]
```

Next, we need to write functions that change velocities. Here we assume that on `X` coordinate, objects maintain its uniform motion, and on `Y`, objects are uniformly accelerated by gravity. Then acceleration functions can be code as follows.

```
# fun accelerateX (p as {Vx:real,...}, t:real) = p;
val accelerateX = fn : ['a#{Vx: real}. 'a * real -> 'a]
# fun accelerateY (p as {Vy:real,...}, t:real) = p # {Vy = Vy + 9.8};
val accelerateY = fn : ['a#{Vy: real}. 'a * real -> 'a]
```

`accelerateX` is constant and therefore not needed in this simple case, but we write one for future refinement.

The function `next` can be obtained by composing all of them below.

```
fun move (p, t) =
  let
    val p = accelerateX (p, t)
    val p = accelerateY (p, t)
    val p = moveX (p, t)
    val p = moveY (p, t)
  in
    p
  end
```

The resulting code is highly modular and type safe, and can be easily refined. For example, if we want to add deacceleration on `X` coordinate by 1% per second, one need only re-write `accelerateY` to the following.

```
# fun accelerateX (p as {Vx:real,...}, t:real) = p # {Vx = Vx * 0.90};
val accelerateX = fn : ['a#{Vx: real}. 'a * real -> 'a]
```

For this `next` function, SML# infers the type we designed at the beginning of this section, and therefore can be applied to any object having many other attributes.

8.6 Representing objects

Records are the fundamental data structures, and they are the basis for various data manipulation models such as relational databases and object-oriented programming. As explained in Chapter12, SML# seamlessly integrate SQL based on record polymorphism. Object-oriented programming is based on a different computation model than functional programming, and we do not hope to represent it in a functional language. However, generic object manipulation underlying object-oriented programming is naturally represented using record polymorphism.

An object has a statue, receives method selector and invoke the designated method on its state. A class can be regarded as the set of methods that belongs to the class. So we can represent a class structure as a record of methods. Each method is coded as a function that takes a state of an object and update it. For example, consider an object in a `pointClass` having `X` coordinate, `Y` coordinate, `Color` property. An object state can be represented by a reference to a record of the those values such as `{X = 1.1, Y = 2.2}`. Then a method can be defined as a function that takes an object state as `self` and update the state. For example, a method to set a value in the `X` coordinate can be coded as follows.

```
# fn self => fn x => self := (!self # {X = x});
val it = fn : ['a#{X: 'b}, 'b. 'a ref -> 'b -> unit]
```

This method can be applied to any objects that contain `X` attributes. A class can be a record consisting of these methods with appropriate names. For example, `pointClass` can be defined as below.

```
val pointClass =
    {
      getX = fn self => #X (!self),
      setX = fn self => fn x => self := (!self # {X = x}),
      getY = fn self => #Y (!self),
      setY = fn self => fn x => self := (!self # {Y = x}),
      getColor = fn self => #Color (!self),
      setColor = fn self => fn x => self := (!self # {Color = x})
    }
```

An object receives a message and invoke the corresponding method on itself. In SML#, this is coded below.

```
local
  val state = ref {X = 0.0, Y = 0.0}
in
  val myPoint = fn method => method pointClass state
end
```

We can similarly define an object having `Color` attribute.

```
local
  val state = ref {X = 0.0, Y = 0.0, Color = "Red"}
in
  val myColorPoint = fn method => method pointClass state
end
```

Under these definition, we can write the following code.

```
# myPoint # setX 1.0;
val it = () : unit
# myPoint # getX;
val it = 1.0 : real
# myColorPoint # getX;
val it = 0.0 : real
# myColorPoint # getColor;
val it = "Red" : string
# myPoint # getColor;
(interactive):15.1-15.12 Error:
  (type inference 007) operator and operand don't agree
  ...
```

As shown in the last example, the system detect all the type errors statically.

8.7 Polymorphic variants

Records are labeled collection of component values. There is a dual concept of this structure, namely *labeled variants*. This may not be familiar to many of you, but the essential ingredients are the same as those of records, so in a system where polymorphic records are supported, polymorphic variants can also be represented. For a type theoretical account, see [9]. In this section, we briefly introduce them through simple examples.

Once you understand records as labeled collections of values, you can think of labeled variants as a value attached with a service request label in a context where a labeled collection of services are defined. For a simple example, let us consider a system where we have two point representations, one in Cartesian coordinates and the other in polar coordinates. In this system, each representation is processed differently, so each point data is attached a label indicating its representation. Polymorphic variant is a mechanisms to those data with polymorphic functions. By regarding the data label as a service selector from a given set of services, this system can be represented by polymorphic records. For example, the following show two representations of the same point.

```
# val myCPoint = fn M => #CPoint M {x = 1.0, y = 1.0};
val myCPoint = fn : ['a#{CPoint: {x: real, y: real} -> 'b}, 'b. 'a -> 'b]
# val myPPoint = fn M => #PPoint M {r = 1.41421356237, theta = 45.0};
val myPPoint = fn : ['a#{PPoint: {r: real, theta: real} -> 'b}, 'b. 'a -> 'b]
```

A variant data with a tag T is considered as an object that receives a method suits, select appropriate suit using T as the selector, and applies the selected method to itself. One you understand this idea, then all you have to do is to write up the set of necessary methods for each tag. For example, a set of methods to compute the distance from the origin of coordinates can be coded as below.

```
val distance =
    {
      CPoint = fn x,y,... => Real.Math.sqrt (x *x + y* y),
      PPoint = fn r, theta,... => r
    };
```

This method suit is invoked by applying an variant object to it.

```
# myCPoint distance;
val it = 1.41421356237 : real
# myPPoint distance;
val it = 1.41421356237 : real
```

In this way, various heterogeneous collections can be processed in type safe way.

Chapter 9

SML# feature: other type system extensions

In addition to record polymorphism, SML# extend the Standard ML type system with the following.

1. rank 1 polymorphism, and
2. first-class overloading.

This chapter briefly introduce them.

9.1 Rank 1 polymorphism

Polymorphic types Standard ML type system can infer are those whose type variables are bound at the top-level. For example, for a function

```
fun f x y = (x, y)
```

the following type is inferred.

```
val f = fn : ['a, 'b. 'a -> 'b -> 'a * 'b]
```

In contrast, SML# infers the following nested polymorphic type

```
# fun f x y = (x, y);
val f = fn : ['a. 'a -> ['b. 'b -> 'a * 'b]]
```

This type can be think of as a type function that receives a type τ through type variable `'a` and return a polymorphic type `['b. 'b -> `τ` * 'b]`. It behaves as follows.

```
# f 1;
val it = fn : ['b. 'b -> int * 'b]
# it "ML";
val it = (1, "ML") : int * string
```

Let t represent type variables, τ monomorphic types (possibly including type variables), and let σ polymorphic types. Then the set of polymorphic types Standard ML can infer is roughly given by the following grammar.

$$\begin{aligned} \tau &::= t \mid b \mid \tau \to \tau \mid \tau * \tau \\ \sigma &::= \tau \mid \forall(t_1, \ldots, t_n).\tau \end{aligned}$$

We call this set rank 0 types.

In contract, SML# can infer the following set called rank 1 types.

$$\begin{aligned} \tau &::= t \mid b \mid \tau \to \tau \mid \tau * \tau \\ \sigma &::= \tau \mid \forall(t_1, \ldots, t_n).\tau \mid \tau \to \sigma \mid \sigma * \sigma \end{aligned}$$

We have introduced this as a technical extension for making compilation of record polymorphism more efficient. In a type system for pure ML term without imperative feature, this extension does not increase the expressive power of ML. However, with the value restriction introduced in the revision of Standard ML type system, rank 1 extension become important extension. Next section explain this issue.

9.2 Value polymorphism restriction and rank 1 typing

It is known that ML's polymorphic typing breaks down when imperative features in Section 7.13 are introduced. Since reference cells can be created for value of any type, one might think that they are polymorphic primitives having the following types.

```
val ref : ['a. 'a -> 'a ref]
val := : ['a. 'a ref * 'a -> unit]
```

However, under these typings, the following incorrect code could be written, which would destroy the system.

```
val idref = ref (fn x => x);
val _ = idref := (fn x => x + 1);
bval _ = !idref "wrong"
```

`['a. ('a -> 'a) ref]` is inferred for `idref`. This type can be used as `(int -> int) ref`, and so does the type of `:=`. So the second line code is accepted. At this point, `idref` is changed to a reference of value of type `int -> int`, but its type remains to be `['a. ('a -> 'a) ref]` and the third line is accepted, and executed. However, this result in applying an integer function to a string.

In order to avoid this problem, Standard ML introduces the restriction:

> polymorphism is restricted to *value expressions*

This is called *value polymorphism restriction*. Value expressions are those that do not execute any computation, such as constants, variables, and function expressions. `ref (fn x => x)` calls primitive `ref` and therefore not a value expression. With this restriction, function applications cannot be given a polymorphic type. In this (rather crude) way, ML prevents a function containing a reference cannot have a polymorphic type, which cause the above problem.

SML# follows this value polymorphism restriction. However, in SML#, polymorphic types can be given to sub-expression such as function body or record component, value restriction is significantly reduced in practice. For example, consider the following function definition and application.

```
val f = fn x => fn y => (x, ref y)
val g = f 1
```

In Standard ML, `f` is given the following type.

```
val f = _ : ['a, 'b. 'a -> 'b -> 'a * 'b ref]
```

Then an application of `f` such as `f 1` contains a free type variable, but since this is not a value expression and therefore the free type variable cannot be rebind to make a polymorphic type. In contrast, SML# infer the following rank 1 type for `f`.

```
val f = _ : ['a. 'a -> ['b. 'b -> 'a * 'b ref]]
```

The function result type `['b. 'b -> 'a * 'b ref]` is inferred before the function application. As a result, the type of `f 1` is obtained by replacing `'a` with `int` in `['b. 'b -> 'a * 'b ref]` without re-inferring a polymorphic type as seen below.

```
val g = _ : ['a. 'b -> int * 'b ref]
```

9.3 First-class overloading

In ML, some of commonly used primitives are overloaded. For example, binary addition `+` can be used on several types included `int, real, word` as shown below.

```
# 1 + 1;
val it = 2 : int
# 1.0 + 1.0;
val it = 2.0 : real
# 0w1 + 0w1;
val it = 0wx2 : word
```

Different from polymorphic functions, a different implementation for `+` (i.e. one of `Int.+, Real.+, Word.+` in the above example) is selected based on the context in which it is used. In Standard ML this overloading is resolved at the top-level at the end of each compilation unit. If there remain multiple possibilities then a predetermine one is selected. For example, if you write

```
fun plus x = x + x
```

in Standard ML, then `Int.+` is selected for `+` and `plus` is bound to a function of type `int -> int`.

This strategy works fine, but this will become a big obstacle in integrating SQL, where most of the primitives are overloaded. If we determine the types of all the primitives in a SQL query at the time of is definition, then we cannot make full use of ML polymorphism in dealing with databases. For this reason, SML# introduces a mechanism to treat overloaded primitive functions as first-class functions. For explain, SML# infers the following polymorphic type for `plus`.

```
# fun plus x = x + x;
val plus = fn : ['a::{int, word, int8, word8, ...}. 'a -> 'a]
```

This function can be used as a function of any type obtained by replacing `'a` with one of `int`, `word`, `int8`, `word8`, `int16`, `word16`, `int64`, `word64`, `intInf`, `real`, `real32` (`int16`,... are omitted). The constraint `'a::{...}` on type variable `'a` indicates the set of allowable instance types.

Chapter 10

SML# feature: direct interface to C

C is the standard language in system programming. SML# support direct interface to C. This feature allows the programmer to use various OS libraries and other low-level functions implemented in C. This chapter explain this feature.

10.1 Declaring and using C functions

In order to use C function, you only have to declare it in SML# in the following syntax.

```
val id = _import "symbol" : type
```

symbol is the name of the C function. *type* is its type. Next section explain how to write the type of a C function.

This declaration instructs SML# compiler to link the named function and bind the SML# variable *id* to that function. A target function linked by this declaration can be any code as far as it is in a standard calling convention of the OS in which SML# runs. The linking to the function is performed at linking time. So, to produce an executable file of SML# program containing this `_import` declaration, it is required to specify either a library or an object file to the command line of SML# command to link it with the SML# program. Some of standard C libraries (including libc and libm in Unix family OS) are linked by default.

This declaration can appear whenever `val` declaration is allied. After this declaration, variable *id* can be used as an ordinary variable defined in SML#.

As an example, consider the standard C library function.

```
int puts(char *);
```

This function takes a string, appends a newline code and outputs it to the standard output, and returns the number of characters actually printed. If printing fails, then it returns the integer representing `EOF` (which is -1 in Linux). This function can be used by writing the following declarations.

```
val puts = _import "puts" : string -> int
```

As seen in this example, C function can bound and used just by writing `_import` keyword followed by the name and the type of the desired function. The following is interactive session using `puts`.

```
# val puts = _import "puts" : string -> int;
val puts = _ : string -> int
# puts "My first call to a C library";
My first call to a C library
val it = 29 : int
# map puts ["I","became","fully","operational","in","April","6th","2012."];
I
became
fully
operational
in
April
```

```
2nd
2012.
val it = [2, 7, 6, 12, 3, 6, 4, 5] : int list
```

The imported C functions can be freely used according to the ML's programming principle – "expressions are freely composed as far as they are type consistent".

10.2 Declaring types of C functions

C functions that can be imported to SML# are those whose types are representable in SML#. The type in `_import` declaration is the type of C function written in SML# notation. The variable specified in the `val` declaration is bound to the imported C function. The type of this variable is an SML# function type that is corresponding to the type of the C function.

A C function type in `_import` declaration is of the form:

`(`τ_1`, `τ_2`, ···, `τ_n`) -> `τ

This notation represents a C function that takes n arguments of type $\tau_1, \tau_2, \ldots, \tau_n$ and returns a value of τ type. Parentheses can be omitted if there is just one argument. What τ is is described below. If there is no argument, write as follows:

`() -> `τ

If the return type is `void` in C, write as follows:

`(`τ_1`, `τ_2`, ···, `τ_n`) -> ()`

If the C function has a variable length argument list, use the following notation:

`(`τ_1`, ···, `τ_m`, ...(`τ_{m+1}`, ···, `τ_n`)) -> `τ

This is the type of a C function that takes at least m arguments followed by variable number of arguments. Since SML# does not support variable length argument lists, user need to specify the number and types of arguments in the variable length part of the argument list. This notation means that arguments of τ_{m+1}, ..., τ_n type are passed to the function.

In the argument list and the return type of the above C function type notation, you can specify any SML# type that is corresponding to a C type. In what follows, we refer to such SML# types as *interoperable types*. The set of interoperable types include the following:

- Any integer type except `IntInf`, such as `int`, `word`, and `char`.
- Any floating-point number type, such as `real` and `Real32.real`.
- Any tuple type whose any field type is an interoperable type, such as `int * real`.
- Any vector, array and ref type whose element type is an interoperable type, such as `string`, `Word8Array.array`, `int ref`.

The correspondence between these interoperable types and C types are given as follows:

- The following table shows the correspondence on integer types and floating-point number types.

SML#'s interoperable type	corresponding C type	note
`char`	`char`	Signedness is not specified, similar to C.
`word8`	`unsigned char`	We assume a byte is 8 bit.
`int`	`int`	Natural size of integers
`word`	`unsigned int`	
`Real32.real`	`float`	IEEE754 32-bit floating-point numbers
`real`	`double`	IEEE754 64-bit floating-point numbers

- Any τ `vector` and τ `array` type is corresponding to the type of a pointer to an array whose element type is the C type corresponding to τ type. The array pointed by an `array` pointer is mutable. In contrast, that of an `vector` pointer is immutable. In other words, the element type of an array pointed by an `vector` pointer is qualified by `const` qualifier.

- `string` type is corresponding to the type of a pointer to an array of `const char`. The last element of the array pointed by a `string` pointer is always terminated by a null character. So a `string` pointer can be regarded as a pointer to a null-terminated string.
- Any τ `ref` type is corresponding to the type of a pointer to the C type corresponding to τ. A `ref` pointer points to a mutable array of just one element.
- Any tuple type τ_1 `*` $\cdots$ `*` τ_n is corresponding to the type of a pointer to a immutable structure whose members are $\tau_1, \ldots, \tau_n$ type in this order. If all of $\tau_1, \ldots, \tau_n$ are same type, this tuple type is also corresponding to the type of a pointer to an array.

Values constructed by SML# are passed transparently to C functions without any modification and conversion. So, user can pass an array allocated in SML# program to a C function that modifies the given array, and obtain the modification by the C function in SML# program.

The entire type of a C function is converted to an SML# function type whose argument type is a tuple type of the argument list types of the C function. There is the following limitation in usage of interoperable types in the C function notation:

- Interoperable types that corresponds to a C pointer type, such as array type and tuple type, cannot be specified as a return type of a C function.

10.3 Basic examples of importing C functions

Let us import some standard C library functions to SML#. At first, we need to look up their prototype declarations from the manual or the header file of the functions we intend to import. Here, suppose we want to import the following functions.

```
double pow(double, double);
void srand(unsigned);
int rand(void);
```

Next, we need to write the types of the above functions by using the interoperable types that corresponds to the argument and return types of those functions.

```
val pow = _import "pow" : (real, real) -> real
val srand = _import "srand" : word -> ()
val rand = _import "rand" : () -> int
```

Then these C functions are imported to SML# as SML# functions of the following types.

```
val pow : real * real -> real
val srand : word -> unit
val rand : unit -> int
```

The `printf` function can also be imported to SML# by using the notation of the variable length argument list. The prototype of `printf` is given below.

```
int printf(const char *, ...);
```

Corresponding to this prototype, we can import `printf` by the following `_import` declaration.

```
val printfIntReal = _import "printf" : (string, ...(int, real)) -> int
```

The type of `printfIntReal` is as follows.

```
val printfIntReal : string * int * real -> int
```

We note that when calling this `printIntReal` function, its first argument must be a output format string that requires just two arguments of an integer and a floating-point number in this order.

Importing a function that takes pointer arguments needs special attention. In C, we often use pointer arguments in two ways; passing a large data structure by a reference to it, or specifying a buffer to store the result of a function. An interoperable type is corresponding to one of the usage of pointer arguments. So we need to carefully choose an interoperable type representing a pointer argument by its usage according to the manual of the C function we intend to import.

Let us import some C functions that have pointer arguments. Suppose we want to import `modf` function. The prototype of `modf` is as follows.

sample.c file:

```
#include <math.h>
double f(const struct {double x; double y;} *s) {
  return sqrt(s->x * s->x + s->y * s->y);
}
```

sample.sml file:

```
val f = _import "f" : real * real -> real
val x = (1.1, 2.2);
val y = f x;
print ("result : " ^ Real.toString y ^ "\n");
```

Execution:

```
# gcc -c sample.c
# smlsharp sample.sml sample.o
# a.out
result : 2.459675
```

Figure 10.1: Passing a tuple to user-defined C function

```
double modf(double, double *);
```

According to the manual of `modf`, the second argument of pointer type must specify the destination buffer of the result of this function. So we specify an interoperable type of a mutable value to the type of the second argument.

```
val modf = _import "modf" : (real, real ref) -> real
```

We need to pay special attention to pointers to `char`. In C, there are a lot of meaning a `char` pointer exactly means. For example, a `char` pointer usually means a null-terminated string. In the other case, a `char` pointer is used as a pointer to most generic type of binary data buffers. Suppose we try to import `sprintf` function. Its prototype is given below.

```
int sprintf(char *, const char *, ...);
```

The first `char` pointer is a pointer to a destination buffer, and the second one represents a null-terminated string. According to this difference of usage of two pointers, we give the following type annotation to the `_import` declaration.

```
val sprintfInt = _import "sprintf" : (char array, string, ...(int)) -> int
```

We can import any user-defined C function to SML#, while we only use standard C library functions to describe how to import C functions so far. Figure 10.3 shows an example of importing an user-defined function to SML# and passing a structure from SML# to the C function.

10.4 Using dynamically linked libraries

The declaration

`val` *id* `= _import "`*symbol*`" :` *type*

is statically resolved to a C function having the name *symbol*. When SML# compiles a source file containing this declaration, the compiler generates an object file containing *symbol* as an external name. When an executable program is build from these objects files, the system linker links these object files with C functions. This is also true in interactive mode, which is implemented by a simple iteration that performs separate compilation, linking and loading.

However for a library that is only available at runtime, such static resolution is impossible. To cope with those situation, SML# provide dynamic linking through the following module.

samle.c file :

```
int f(int s) {
  return(s * 2);
}
```

Execution:

```
$ gcc -shared -o sample.so sample.c
$ smlsharp
SML# version 1.00 (2012-04-02 JST) for x86-linux
# val lib = DynamicLink.dlopen "sample.so";
val lib = _ : lib
# val fptr = DynamicLink.dlsym(lib, "f");
val fptr = ptr : unit ptr
# val f = fptr : _import int -> int;
val f = _ : int -> int
# f 3;
val it = 6 : int
```

Figure 10.2: Using dynamic link library

```
structure DynamicLink : sig
  type lib
  type codeptr
  datatype scope = LOCAL | GLOBAL
  datatype mode = LAZY | NOW
  val dlopen : string -> lib
  val dlopen' : string * scope * mode -> lib
  val dlsym : lib * string -> codeptr
  val dlclose : lib -> unit
end
```

These functions provide the system services of the same names provide in a Unix-family OS.

- `dlopen` opens a shared library.
- `dlopen'` takes one of those parameters to control `dlopen`. `RTLD_LOCAL`, `RTLD_GLOBAL` and `RTLD_LAZY`, `RTLD_NOW`. For its details, consult OS manual on `dlopen`.
- `dlsym` takes a library handle obtained by `dlopen` and a function name, and returns a C pointer to the function.
- `dlclose` closes the shared library.

The C pointer returned by `dlsym` can be converted to SML# function by the following expression.

exp `: _import` *type*

exp is a SML# expression of type `codeptr`. *type* specifies the type of C function as in `_import` declaration. Figure 10.2 shows an example.

Chapter 11

SML# feature: Multithread programming

Through the seamless and direct C interface, SML# supports multiple threads that run concurrently on multicore processors. SML# allows you to exploit two thread libraries directly in SML#: POSIX thread (Pthread) library, which is a part of operating systems; and MassiveThreads, a lightweight fine-grain thread library. In both libraries, you can create multiple threads in SML#, each of which executes a SML# routine and is scheduled to a CPU core by operating systems or the MassiveThreads' thread scheduler. This chapter introduces multithread programming in SML#.

11.1 Programming with Pthreads

SML# provides the `Pthread` structure that is a direct binding of the Pthread library in SML#. The `pthread_create` and `pthread_join` functions, which creates and join a thread, respectively, are provided as the following SML# functions:

```
Pthread.Thread.create : (unit -> int) -> Pthread.thread
Pthread.Thread.join : Pthread.thread -> int
```

The following is a simple example that computes `fib 42` in a different thread.

```
# fun fib 0 = 0 | fib 1 = 1 | fib n = fib (n - 1) + fib (n - 2);
val fib = fn : int -> int
# val t = Pthread.Thread.create (fn _ => fib 42);
val t = ptr : Pthread.thread
# Pthread.Thread.join t;
val it = 267914296 : int
```

The SML# runtime system for direct C interface is carefully designed so that SML# functions can be passed to a C function as call-back functions and can be called back from a C function running in a thread that is different from the one that originally calls the C function. Using this features, the programmer can enjoy multi-thread programming simply by importing the Pthread library. To do this, we define the type

```
type pthread_t = unit ptr
```

for Pthread handles. τ `ptr` is a built-in type for C pointers; `unit ptr` corresponds to `void*` type in C.

The thread creation function, `pthread_create`, is then imported by the following declaration.

```
val pthread_create =
    _import "pthread_create"
    : (pthread_t ref, unit ptr, unit ptr -> unit ptr, unit ptr) -> int
```

Using this, we can write a function `spawn` that creates a thread as follows.

```
fun spawn f =
    let
      val r = ref (Pointer.NULL ())
    in
      pthread_create (r,
                      Pointer.NULL (),
                      fn _ => (f () : unit; Pointer.NULL ()),
                      Pointer.NULL ());
      !r
    end
```

`Pointer.NULL ()` return s the `NULL` pointer in C.

Similarly, `pthread_join` is imported and used to define a function `join` that waits for the termination of a created thread.

```
val pthread_join =
    _import "pthread_join"
    : (pthread_t, unit ptr ref) -> int
fun join t =
    (pthread_join (t, ref (Pointer.NULL ())); ())
```

Using these functions, we can write the same program as shown at the beginning of this section.

```
# fun fib 0 = 0 | fib 1 = 1 | fib n = fib (n - 1) + fib (n - 2);
val fib = fn : int -> int
# val r = ref 0;
val r = ref 0 : int ref
# fun g () = r := fib 42;
val g = unit -> unit
# val t = spawn g;
val t = ptr : unit ptr
# join t;
val it = () : unit
# !r;
val it = 267914296 : int
```

Note that it is dengerous to replace the line

```
# val t = spawn g;
```

with

```
# val t = spawn (fn () => r := fib 42);
```

because the garbage collector may collect the closure generated by the function expression before the function is called in another thread. See Section 29.2 for details.

11.2 Fine-grain multithread programming with MassiveThreads

SML# provides MassiveThreads-based multithread support by default. MassiveThreads is a user-level lightweight thread library in C provided by the University of Tokyo. The SML#'s direct C interface and unobtrusive concurrent garbage collection enable SML# programs to call MassiveThreads directly. MassiveThreads allows us to create millions of user threads, say 1,000,000 threads, that runs concurrently on multicore processors.

By default, time SML# runtime is restricted to use only one CPU core for the performance of single-thread programs. To enable MassiveThreads on multicore processors, specify at least one `MYTH_*` environment variable as a configuration of MassiveThreads. A typical one is `MYTH_NUM_WORKERS` that specifies the number of worker threads, i.e., the number of CPU cores the program uses. For example, do the following command to start an interactive session:

```
$ MYTH_NUM_WORKERS=0 smlsharp
```

In Linux, setting `MYTH_NUM_WORKERS` to 0 means using all available CPU cores.

SML# provides `Myth` structure that is a direct binding of MassiveThreads library in SML#. In `Myth.Thread` structure, you find basic functions for thread management. Its primary functions are the following:

- User thread creation.

  ```
  Myth.Thread.create : (unit -> int) -> Myth.thread
  ```

  `create` f creates a new user thread that computes f `()`. The created user thread will be scheduled to an appropriate CPU core by the MassiveThreads task scheduler. Its scheduling policy is non-preemptive; a thread occupies a CPU core until either it calls a thread control function of MassiveThreads (`Myth.Thread.yield`) or it terminates.

- User thread join.

  ```
  Myth.Thread.join : Myth.thread -> int
  ```

  `join` t waits for the completion of thread t and returns the result of the computation of t. Any user thread created by `create` must be `join`ed sometime in the future. Note that this `Myth.Thread` structure is just a direct binding of C functions, as in C, the resource of the created threads must be freed explicitly.

- User thread scheduling.

  ```
  Myth.Thread.yield : unit -> unit
  ```

  `yield ()` yields the CPU core to other threads.

As an introduction to MassiveThreads programming, let us write a task parallel program. Roughly speaking, you can write a task parallel program by the following steps:

1. Write a recursive function that performs divide-and-conquer.
2. Surround each recursive call with a pair of `create` and `join` so that each recursive call is evaluated in a different thread.
3. To prevent from creating very short threads, set a threshold (cut-off) to stop thread creation and do recursive calls in the same thread. The threshold must be decided so that sequential wall-clock time of a user thread is sufficiently longer than the overhead of thread creation. In practice, 3–4 microseconds for a user thread is good enough.

For example, let us write a program that compute `fib 40` recursively. The following is a typical definition of recursive `fib` function:

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n - 1) + fib (n - 2)
val result = fib 40
```

To compute `fib (n - 1)` and `fib (n - 2)` in parallel, surround one of them with `create` and `join`:

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n =
    let
      val t2 = Myth.Thread.create (fn () => fib (n - 2))
    in
      fib (n - 1) + Myth.Thread.join t2
    end
val result = fib 40
```

This is not a goal; unfortunately, if `n` is very small, the computation cost of `fib n` is apparently much smaller than the overhead of thread creation. To avoid this, we introduce a threshold so that it computes sequentially if `n` is smaller than 10.

```
val cutOff = 10
fun fib 0 = 0
  | fib 1 = 1
  | fib n =
    if n < cutOff
    then fib (n - 1) + fib (n - 2)
    else
      let
        val t2 = Myth.Thread.create (fn () => fib (n - 2))
      in
        fib (n - 1) + Myth.Thread.join t2
      end
val result = fib 40
```

Now it is all done! Running this program, it eventually generates 3,524,577 user threads in total.

Chapter 12

SML# feature: seamless SQL integration

Accessing databases is essential in most of practical programs that manipulate data. The most widely used database query language is SQL. The conventional method of accessing databases to generate SQL command string, which is cumbersome and error prone. SML# integrates SQL expressions themselves as polymorphically typed first-class citizens. This chapter explain this feature.

12.1 Relational databases and SQL

Most of practical database systems are relational databases. To understand SML# database integration, this section review the basics notions of relational databases and SQL.

In the relational model, data are represented by a set of relations. A relation is a set of tuples, each of which represents association of attribute values such as name, age, and salary. Such a relation is displayed as a table of the following form.

name	age	salary
"Joe"	21	10000
"Sue"	31	20000
"Bob"	41	20000

A relational database is system to manipulate a collection of such tables. A relation R on the sets $A_1, A_2, \cdots, A_n$ of attribute values is mathematically a subset of the Cartesian product $A_1 \times A_2 \cdots \times A_n$. Each element t in R is an n element tuple $(a_1, \ldots, a_n)$. In an actual database system, each component of a tuple has attribute name, and a tuple is represented as a labeled record. For example, the first line of the example table above is regarded as a record {`name="Joe", age=21, salary=1000`}. On these relations, a family of operations are defined, including union, projection, selection, and Cartesian product. A set of tables associated with a set of these operations is called the relational algebra. One important thing to note on this model is that, as its name indicates, the relational model is an algebra and that it is manipulated by an algebraic language. An algebraic language is a functional language that does not have function expression.

In relational databases, the relational algebra is represented by the language called SQL, which is language of set-value expressions. The central construct of SQL is the following SELECT expression.

```
SELECT t1.l1 as l'1,..., tm.lm as l'm
FROM R1 as t1, ..., Rn as tn
WHERE P(t1,...,tn)
```

Here we used the following meta variables.

- R: relation variables
- t: tuple variables
- l: labels, or attribute names

- $t.l$: the l attribute of tuple l

The operational meaning of a SELECT expression can be understood as follows.

1. Evaluate each R_i in `FROM` clause, and generate their Cartesian product $R_1 \times \cdots \times R_n$
2. Let $(t_1, \ldots, t_n)$ be any representative tuple in the product.
3. Select the tuples that satisfies the predicate $P(t_1, \ldots, t_n)$ specified in `WHERE` clause from the product.
4. For each element $(t_1, \ldots, t_n)$ in the select set, construct a record $\{l'_1$=$t^1.l_1$, ..., l'_m=$t^m.l_m\}$.
5. Collect all these records.

For example, let the above example table be named as `Persons` and consider the following SQL.

```
SELECT P.name as name, P.age as age
FROM Persons as P
WHERE P.salary > 10000
```

This expression is evaluated as follows.

- The Cartesian product of the soul relation `Persons` is `Persons` itself.
- Let `P` be any tuple in `Persons`.
- Select from `Person` all the tuples `P` such that `P.Salary > 10000`. We obtain the following set.

name	age	salary
"Sue"	31	20000
"Bob"	41	20000

- For each tuple `P` in this set, compute the new tuple {`name=P.name, age=P.age`} to obtain the following set.

name	age
"Sue"	31
"Bob"	41

  The is the result of the expression.

  This result represent the set (list) of records: {{`name="Sue", age=31`}, {`name="Bob", age=31`}}.

12.2 Integrating SQL in SML#

We observe that SQL `SELECT` command is an expression that construct a table from a set of tables specified in `FROM` clause. In practice, SQL commands are usually evaluated against a particular database connection. Since database connection is conceptually a set of tables, if we generalize SQL as a language that takes a database connection and returns a relation, an SQL command can be regarded as a function that takes a set of tables and returns a table. As we have already observe that SQL `SELECT` is an algebraic expression. Then SQL is a functional language on tables.

Since tables are set of records, table value functions should be typable using record polymorphism. According to our analysis, there is one subtle point in typing beyond record polymorphism [14]. For this reason, SML# introduced the following special syntax for SQL query functions.

```
_sql db => select #P.name as name, #P.age as age
           from #db.Persons as P
           where #P.salary > 10000
```

`_sql db => ...` represent a function that takes a database connection through parameter `db`. `>` in a `_sql` expression is SQL primitive for numerical comparison operation defined in the `SQL` module that implement various SQL specific primitives. `#P.Name` selects `Name` field from `P`. `#db.Persons` selects `Persons` table from `db`. These correspond to `#Name P` and `#Persons db` in SML# expression. In `_sql` x `=>` *expr* expression, we choose these syntax that are closer to SQL SELECT commands.

For this expression, SML# infers the following polymorphic type.

```
val it = fn
   : ['a#{Persons: 'b list},
     'b#{age: 'c, name: 'e, salary: 'g},
     'c::{int, intInf, word, char,...},
     'd::{int, intInf, word, char,...},
     'e::{int, intInf, word, char,...},
     'f::{int, intInf, word, char,...},
     'g::{int, intInf, 'h option},
     'h::{int, intInf}.
       'a SQL.conn -> {age: 'c, name: 'e} SQL.cursor]
```

This type indicates that this query is a function from a database connection of type `'a conn` to a table of tuple `{age: 'c, name: 'e}`. `'a` represents the structure of the input database.

This is indeed a most general polymorphic type of the above query, By the fact that SML# can infer a most general polymorphic type, we are guarantees that SQL can be used in SML# based on ML programming principle: "expressions are freely composed as far as they are type consistent", i.e. SQL expressions can be freely combined with any other language construct of SML# as far as they are type consistent. This will open up flexible and type safe programming using databases.

12.3 Query execution

Database can be accessed by applying query function defined by `_sql` x `=>` exp expression to a database connection. For this purpose, SML# provides the following constructs.

- Database server expressions

  `_sqlserver` *serverLocation* `:` τ

  This expression locates a database server. *serverLocation* is the location information of a database server. Its concrete syntax is determined by the database system to be connected. τ is the type of the database to be connected. It describes the table names and their types using the syntax of record types. Evaluating this expression always succeeds and yields a database server object of type τ `SQL.server`, which contains the database server information. See Section 22.3 for details.

- Database connection primitive.

  ```
  SQL.connect : ['a. 'a SQL.server -> 'a SQL.conn]
  ```

  This primitive takes a database server value, extract the database location stored in the value, attempts to connect to the database, and if successful it checks that the connected database indeed has the structure described by τ, and then return a database connection object of type τ `SQL.conn`.

- Database query execution. The `_sql` expression itself is a function that executes its corresponding query on a database. By applying a connection to a query function, the query is executed on a database through given connection. The result is returned as a cursor of type τ `SQL.cursor`.

- Conversion of the query result.

  ```
  SQL.fetchAll : ['a. 'a SQL.cursor -> 'a list]
  SQL.fetch : ['a. 'a SQL.cursor -> ('a * 'a SQL.rel) option]
  ```

  `SQL.fetch` fetches the first tuple at the cursor and forward the cursor to the next tuple. `SQL.fetchAll` reads all tuples after the cursor and converts them to a list of records.

- Post processing.

  ```
  SQL.closeCursor : ['a. 'a SQL.cursor -> unit]
  SQL.closeConn : ['a. 'a SQL.conn -> unit]
  ```

  `SQL.closeCursor` terminates the query and `SQL.closeConn` close a database connection.

12.4 Query examples

Let us now connect and use an actual database from SML#. SML# version 4.2.0 support PostgreSQL. To use a database, you need to install and start PostgreSQL server.

Here we show a standard step in Linux system to set up PostgreSQL server. For more details, consult PostgreSQL document.

1. Login as postgres.
2. Start PostgreSQL server. The command `pg_ctl start -D /usr/local/pgsql/data` will do this.
3. Create a PostgreSQL user role by executing command `createuser` *myAccount*, where *myAccount* is your user account.
4. Return to your account, and execute command `createdb mydb` to create a database called mydb.
5. Use the SQL interpreter to create a table. For example, the table in Section 12.1 can be created as follows.

```
$ psql mydb
mydb# CREATE TABLE Persons (
    name text not null, age int not null, salary int not null   );
mydb# INSERT INTO Persons VALUES ('Joe', 21, 10000);
mydb# INSERT INTO Persons VALUES ('Sue', 31, 20000);
mydb# INSERT INTO Persons VALUES ('Bob', 41, 30000);
```

6. Check that the database can be accessed. You should get the following output.

```
$ psql mydb;

mydb=# SELECT * FROM Persons;
 name | age | salary
------+-----+--------
 Joe  |  21 |  10000
 Sue  |  31 |  20000
 Bob  |  41 |  20000
(3 rows)
```

Now let's access this database from SML#. Let the query function defined in Section 12.2 `myQuery`. In an interactive session, you should get the following.

```
$ smlsharp
# val myServer = _sqlserver SQL.postgresql "dbname=mydb" : {Persons:{name:string,
age:int, salary :int} list};
val myServer = _ : {Persons: {age: int, name: string, salary: int} list} SQL.server
# val conn = SQL.connet myServer;
val conn = _ : {Persons: {age: int, name: string, salary: int} list} SQL.conn
# val rel = myQuery conn;
val rel = {Persons: {age: int, name: string, salary: int}} list SQL.conn
# SQL.fetchAll rel;
val it = {{age=32, name="Sue"}, {age=41, name="Bob"}} : {age:int, name: string}
list
```

12.5 Other SQL statements

SQL contains many other functionalities. SML# version 4.2.0 support the following commands:

- query (`SELECT`),
- insert and delete (`INSERT`, `DELETE`),
- table update (`UPDATE`), and

- transaction (`BEGIN`, `ROLLBACK`, `COMMIT`).

In SELECT queries, the following features are available in addition to the basic features:

- natural join (`NATURAL JOIN`),
- inner join (`INNER JOIN`),
- grouping (`GROUP BY`, `HAVING`),
- subqueries including corelated subqueries and EXISTS subqueries,
- sorting (`ORDER BY`), and
- limitation of row numbers (`LIMIT`, `OFFSET`, `FETCH`)

See Chapter 22 for the syntax of SQL expressions. SML# development team has been working on adding SQL features towards complete integration of SQL in SML#.

Chapter 13

SML# feature: dynamic types and typed manipulation of JSON

In network communication or file I/O, data are serialized to a character string in a specified format. For example, JSON is one of the most popular data serialization format in the Internet. Unlike ML data, serialized data are inherently untyped and heterogeneous. Therefore, to deal with serialized data in ML, untyped programming is needed even if the serialized data have a structure similar to some data structure in ML, such as a record. SML#'s dynamic typing mechanism allows the users to convert between dynamically-structured data including serialized data and statically-typed values. On top of this, SML# provides a type-safe way to manipulate JSON data; it gives the structures of JSON data static types like ML records and provides statically-typed constructs for JSON manipulation. This chapter describes these features.

13.1 Dynamic typing

In dynamically-typed languages, the users are allowed to perform operations depending on the types of values at runtime. A value in a dynamically-typed language is a uniform data structure consisting of the runtime representation of a type and a value of the type. SML# provides a mechanism to deal with this dynamically-typed data structures.

The `Dynamic.void Dynamic.dyn` type is the type of dynamically-typed values. The meaning of `Dynamic.void` shall be described below. The following primitives are given for this type:

- The function `Dynamic.dynamic : ['a#reify. 'a -> Dynamic.void Dynamic.dyn]`. This introduces a dynamically-typed value from an arbitrary ML value. `'a#reify` indicates that the runtime type information of the instance of `'a` is required. Unlike other type kinds, the `reify` kind does not restrict the set of types over which the type variable ranges.
- The expression `_dynamic` *exp* `as` τ. This checks the type information of dynamically-typed value *exp* at runtime and casts it to the type τ. If the cast failed, the runtime exception `Dynamic.RuntimeTypeError` is raised.

Furthermore, the following expression is provided:

`_dynamiccase` *exp* `of` $dpat_1$ `=>` exp_1 `|` $\cdots$ `|` $dpat_n$ `=>` exp_n

This performs dynamic type check and pattern matching with a dynamically-typed value *exp*. A pattern $dpat_i$ may be an arbitrary ML pattern except that variable patterns must be type-annotated.

The following interactive session is an example of a heterogenous list and functions proceeding it.

```
# val x = Dynamic.dynamic {name = "Sendai", wind = 7.6};
val x = _ : Dynamic.void Dynamic.dyn
# val y = Dynamic.dynamic {name = "Shiroishi", weather = "Sunny"};
val y = _ : Dynamic.void Dynamic.dyn
# val z = Dynamic.dynamic {name = "Ishinomaki", temp = 12.4};
val z = _ : Dynamic.void Dynamic.dyn
# val l = [x, y, z];
```

```
val l = [_, _, _] : Dynamic.void Dynamic.dyn list
# fun getName r = _dynamiccase r of {name:string, ...} => name;
val getName = fn : ['a. 'a Dynamic.dyn -> string]
# map getName l;
val it = ["Sendai", "Shiroishi", "Ishinomaki"] : string list
# fun getTemp r =
>     _dynamiccase r of
>       {temp:real, ...} => SOME temp
>     | _ : Dynamic.void Dynamic.dyn => NONE;
val getTemp = fn : ['a. 'a Dynamic.dyn -> real option]
# map getTemp l;
val it = [NONE, NONE, SOME 12.4] : real option list
```

Dynamically-typed values are not restricted to records; values of arbitrary types, such as datatypes, functions, and opaque types, can become dynamically-typed values and get back to statically-typed values.

13.2 Reification of terms and types

In statically-typed langauges, type information is usually meta information that only the compiler manages and therefore it is not included as a data structure in the object code. The memory-level data structure of a value is also meta information depending on the type information. "Reification" is the operation that extracts such meta information to objects that the code can deal with. The SML#'s dynamic typing feature is constructed on top of the reification mechanism. The users is also allowed to access the reification mechanism and obtain type information and internal structure of values as ordinary ML datatype.

SML# provides the following functions for this purpose:

- `Dynamic.dynamicToTerm : Dynamic.void Dynamic.dyn -> Dynamic.term`. This extracts the value structure of a dynamically-typed value as a term representation of the `Dynamic.term` type. The `Dynamic.term` type is an ordinary ML datatype and therefore you can analyze it by the `case` expression.

- `Dynamic.dynamicToTy : Dynamic.void dyn -> Dynamic.ty`. This extracts the type information of a dynamically-typed value as a term representation of the `Dynamic.ty` type. Similarly to `Dynamic.term`, the `Dynamic.ty` type is an ordinary ML datatype.

- `Dynamic.termToDynamic : Dynamic.term -> Dynamic.void Dynamic.dyn`. This constructs a dynamically-typed value from its term representation. By combining `_dynamic` construct, you can convert `Dynamic.term` structures to statically-typed ML values.

The following interactive session is an example of constructing an ML record of different type from another ML record.

```
# open Dynamic;
...
# val x = {name = "Sendai", wind = 7.6};
val x = {name = "Sendai", wind = 7.6} : {name: string, wind: real}
# val d = dynamicToTerm (dynamic x);
val d = RECORD {#name => STRING "Sendai", #wind => REAL64 7.6} : term
# case d of
>   RECORD m =>
>   RECORD (RecordLabel.Map.insert
>             (m, RecordLabel.fromString "weather", STRING "cloudy"))
>   | x => x;
val it =
  RECORD
  {#name => STRING "Sendai", #weather => STRING "cloudy", #wind => REAL64 7.6}
  : term
# termToDynamic it;
```

```
val it = _ : void dyn;
# _dynamic it as {name:string, wind:real, weather:string};
val it =
  {name = "Sendai", weather = "cloudy", wind = 7.6}
  : {name: string, weather: string, wind: real}
```

13.3 Pretty printer

In the SML#'s interactive session, ML values of arbitrary types are printed to the standard output. This feature is realized on top of the reification mechanism. The user is allowed to invoke the printer of the interactive session from user programs. The following functions are available:

- `Dynamic.pp : ['a#reify. 'a -> unit]`. This pretty-prints the given ML value of arbitrary type to the standard output. The format is similar to the interactive session. This is useful for print debug.

- `Dynamic.format : ['a#reify. 'a -> string]`. Instead of printing to standard output, this returns a string of the pretty-printed ML value.

These functions can be used in polymorphic functions. Note that using these functions in a polymorphic function may change the type of the function by adding `reify` kind to the type of the function.

13.4 JSON as a partially dynamic record

JSON is a data structure consisting of the following: basic types such as integers and strings; object types, which associate values with labels; and array types which represent data sequences. These structures would corresponds to basic types, record types, and list types of ML.

However, JSON and ML are inherently different since JSON is untyped and heterogeneous. In JSON, objects received from the same source do not always have the same type of values in a same label. Even the existence of a label is not guaranteed. In addition, all elements in a JSON array does not always have same type. In practice of JSON handling, we often meet such heterogeneous data. This means the two facts: it is difficult to give ML types to JSON data, and it is useless to limit JSON data in those that have ML types.

SML# deals with such heterogeneous JSON data in a type-safe way based on the idea of "partially dynamic records." A partially dynamic record is a dynamically typed records, but some parts of it is statically known. In SML#, every JSON data is regarded as a partially dynamic record.

Let us look into JSON handling in SML# through an example.

When a JSON is read, it initially has `Dynamic.void Dynamic.dyn` type. This type means that it is a JSON whose structure is not statically known at all. For example, suppose that we intend to read the following JSON:

```
{"name" : "Sendai", "wind" : {"speed" : 7.6, "deg" : 170.0}}
```

Its type is `Dynamic.void Dynamic.dyn` at first.

Suppose also that we expect the given JSON includes at least `name` field of string (if it is not found, a runtime exception is raised) and intend to read the value of that field. To do this, we perform a dynamic type check against the JSON to make sure that it has the `name` field of string type. If this check is passed, the JSON certainly has the `name` field as we expected. To represent this fact in static typing, SML# gives the result of this check a type `{name : string} Dynamic.dyn`.

After the dynamic check, we extract its statically known part by using the following function:

```
Dynamic.view : ['a#reify. 'a Dynamic.dyn -> 'a]
```

This function allows us to obtain a record of type `{name : string}` from the JSON of type `{name : string} Dynamic.dyn`. The meaning of type kind `#reify` shall be described below.

We successfully convert the given JSON data to an ML record. Succeeding data processing can be carried out by freely combining ML constructs in a type-safe way.

13.5 Language constructs for JSON manipulation

SML# provides the following constructs:

- Importing a JSON data.

  ```
  Dynamic.fromJson : string -> Dynamic.void Dynamic.dyn
  ```

  This function parses a JSON string and read it as a partially dynamic record. It raises the `Dynamic.RuntimeTypeError` exception if the parsing failed. In the return type, `Dynamic.void` denotes that the structure of the value is not yet statically known.

- Dynamic type checking expression `_dynamic` *exp* `as` τ and dynamic patttern matching expression `_dynamiccase`. These expressions described in Section 13.1 are also useful for JSON type checking. The following types are allowed as τ:

  - Ordinary ML types such as basic types such as `int`, `bool`, and `string`; record types; list types; and arbitrary combination of these. Only if the structure of the given JSON matches with τ completely, the type cast succeeds.
  - Partially dynamic record types of the form $\{l_1$`:`τ_1`, ` $\cdots$`, ` l_n`:`$\tau_n\}$ `Dynamic.dyn`. Only if the given JSON is a object that have at least labels $l_1, \ldots, l_n$ and the value of each label l_i can be casted to τ_i, then the type cast succeeds.
  - Completely dynamic type `Dynamic.void Dynamic.dyn`. The type cast to this type always succeeds.
  - Arbitrary nested combination of the above types.

- Obtaining a static view.

  ```
  Dynamic.view : ['a#reify. 'a Dynamic.dyn -> 'a]
  ```

  This function converts the statically known part of the given partially dynamic record to its corresponding ML data structure. It raises the `Dynamic.RuntimeTypeError` exception if the argument is of type `Dynamic.void Dynamic.dyn`.

- JSON printer.

  ```
  Dynamic.toJson : ['a. 'a Dynamic.dyn -> string]
  ```

  This function returns the string representation of the given JSON. The result includes all fields in the given JSON regardless of the type instance of `'a`.

As seen in the type of JSON data, it is allowed to use functions for dynamically-typed values and reification with JSON data. Conversely, it is also allowed to apply these functions for partially dynamic records to dynamically-typed values.

13.6 Examples of JSON programming

The following interactive session is an example of JSON program that reads a list of records written in JSON.

```
# val J = "[{n̈ame:J̈oe, äge:21, g̈rade:1.1},"
            ^ "{n̈ame:S̈ue, äge:31, g̈rade:2.0},"
            ^ "{n̈ame:B̈ob, äge:41, g̈rade:3.9}]";
val J =
  "[{"name":"Joe", "age":21, "grade":1.1},
    {"name":"Sue", "age":31, "grade":2.0},
    {"name":"Bob", "age":41, "grade":3.9}]" : string
# fun getNames l = map #name l;
val getNames = fn : ['a#{name: 'b}, 'b. 'a list -> 'b list]
# val j = Dynamic.fromJson J;
val j = _ : Dynamic.void Dynamic.dyn
```

```
# val vl = _dynamic j as {name:string, age:int, grade:real} list;
val vl =
  [
   {age = 21,grade = 1.1,name = "Joe"},
   {age = 31,grade = 2.0,name = "Sue"},
   {age = 41,grade = 3.9,name = "Bob"}
  ] : {age: int, grade: real, name: string} list
# val nl = getNames vl;
val nl = ["Joe","Sue","Bob"] : string list
```

Let us look at a more practical example. Suppose that we intend to deal with the following JSON.

```
[
  {"name":"Alice", "age":10, "nickname":"Allie"},
  {"name":"Dinah", "age":3, "grade":2.0},
  {"name":"Puppy", "age":7}
]
```

Let `J` be the JSON string of this. This is a heterogeneous list. While it is not able to cast such a list to an ML list, we can see that every element in this list has at least `name` and `age` field of string and int type, respectively. According to this view, we read this JSON as follows:

```
val j = Dynamic.fromJson J
val vl = _dynamic j as {name:string, age:int} Dynamic.dyn list
```

The type of `vl` is `{name:string, age:int} JSON.dyn list`, which means a list of partially dynamic records. The static view of this list can be obtained by

```
val l = map Dynamic.view vl
```

Then, to obtain `name` values from the elements, for example, do the following:

```
val names = map #name l
```

We usually want to check whether or not a certain field exists in a partially dynamic record in the list. We do this by performing the dynamic type check against each element in the list. For example, suppose we want to get `nickname` if it exists, or get `name` otherwise. The following code achieve this:

```
fun getFriendlyName x =
    _dynamiccase x of
      {nickname = y:string, ...} => y
    | _ : Dynamic.void Dynamic.dyn => #name (Dynamic.view x) val friendlyNames
= map getFriendlyName vl
```

The type of `getFriendlyName` is `['a#reify#{name : string}. 'a Dynamic.dyn -> string]`. Note that the combination of a polymorphic record type and partially dynamic type constitutes this type. What does this type mean? Intuitively, from the perspective of JSON manipulation, one would say that this seems similar to `{name : string} Dynamic.dyn -> string`. Strictly speaking, the meaning of these two types are different. Latter one can take a JSON object only if exactly its `name` field is known and thus any other fields must be unknown without further dynamic type checking. In contrast, first one accepts JSON objects if at least its `name` field is known in spite of determination of any other fields.

Chapter 14

SML# feature: separate compilation

One major feature of SML# as a practical language is the support of true separate compilation that is compatible with system development in C. In SML#, one can develop a large software system as follows.

1. Compile source files into object files.
2. Link object files and additional C object files and system libraries to generate an executable file.

Furthermore, SML# compiler can produce file dependency in the format of `Makefile`. Using these features, SML# can be used with C to develop a large software efficiently and reliably. This chapter outline how to use separate compilation of SML#.

14.1 Separate compilation overview

A standard step of separate compilation consists of the following steps.

1. Decompose the system into a set of compilation units. Each component cab be of any size and can contain any sequence of SML# declarations. Here we suppose that we decompose the system into `part1` and `part2`.
2. For each decomposed component, define its interface. The interface is described in the interface language we shall explain in Section 14.3. In our scenario of decomposing the system into `part1` and `part2`, we first define their interface files `part1.smi` and `part2.smi`.
3. For each interface file, develop a source file that implements it. Each source file is independently compiled to an object file.

   In the above scenario, develop `part1.sml` and `part2.sml` for `part1.smi` and `part2.smi`, compile them to generate `part1.o` and `part2.o`. Developing each of the source files and its compilation can be done independently of the other source file. For example, even if `part2` uses some functions in `part1`, `part2.sml` can be compiled before writing `part1.sml`. In ML, type error detection during compilation plays an important role in source file development. So the ability to separately compiling each source file independently of any other source files are important feature in large software development.
4. Link the set of object files with necessary libraries and external object files to generate an executable file. For example, if `part1` and `part2` uses C files and libraries, then link `part1.o` and `part2.o` with those compiled C files and libraries to generate an executable file.

A more advanced scenario can be the following.

1. Decompose the system into the part to be written in C and the part to be written in SML#.
2. For the C part, develop header files (.h files) and source files (.c files).
3. For the SML# part, develop interface files (.smi files) and source files (.sml files).
4. Create `Makefile` using the dependency analysis of SML#.

5. Do `make` to compile and link the necessary files.

The SML# system itself is a large project including a few tools, all of which are written in C and SML#, and developed in this scenario.

14.2 Separate compilation example

Following the simple scenario in the previous section, let us develop an Omikuji (Japanese written oracle drawing) program as an example. Instead of asking Japanese sacred split, we use extremely good random number generator. The system is divided into

- `random`: a random number generator, and
- `main`: the main part.

The fist step is to design interface files as follows.

- `random.smi`:

```
structure Random =
struct
  val intit : int -> unit
  val genrand : unit -> int
end
```

  This interface file says that the implementing source file provide `Random` structure without using any other files.

- `main.smi`:

```
_require "basis.smi"
_require "./random.smi"
```

  This interface file says that it uses `"basis.smi"` (The Standard ML Basis Library) and `random.smi`, and provide no resource.

Using these interface files, `main.sml` and `random.sml` are developed independently. `main.sml` can be given as in Figure 14.1. This file can be compiled without any source file that implements `random.smi`, and checks syntax and type errors by typing:

```
$ smlsharp -c main.sml
```

The `-c` switch instructs SML# compiler to compile the specified source file to an object file. The compiler checks its syntax and types and if no error is detected then it produces `main.o` file. `main.smi` is chosen as its interface file by default. A specific interface file can be specified by including the directive `_interface` *filePath* at the beginning of the source file.

Next, we develop `random.sml`. Development of a high quality random number generator requires expert knowledge in algebraic number theory and careful coding. Here, instead of developing one from scratch, we try to find some existing quality code. Among a large number of implementations, perhaps Mersenne Twister is the best in its quality and efficiency. So we decide to use this algorithm, which is available as a C source file `mt19937ar.c`.

Let us obtain the C source from the Internet by searching for Mersenne Twister or `mt19937ar.c`. The file contains the following function definitions.

```
void init_genrand(unsigned long s);
void init_by_array(unsigned long init_key[], int key_length);
unsigned long genrand_int32(void);
long genrand_int31(void);
double genrand_real1(void);
double genrand_real2(void);
double genrand_real3(void);
double genrand_res53(void);
int main(void)
```

Among them, `main` is a main function that test the algorithm. Since we are defining our executable, the main function should be in the compiled object file of our top level source file `main.sml`. So we comment out `int main(void)` function in the file. `mt19937ar.c`. All the others functions can be used from our program. Here we decide to use the following two.

- `void init_genrand(unsigned long s)` for initializing the algorithm with the seed length `s`, which can be any non-negative integer.
- `long genrand_int31(void)` for generating 31 bit unsigned (i.e. 32 bit non-negative) random number sequence.

The `random.sml` can then be defined as the following code that simply call these functions.

```
structure Random =
struct
  val init = _import "init_genrand" : int -> unit
  val genrand = _import "genrand_int31" : unit -> int
end
```

This source file can be compiled independently of other files by the following command.

```
$ smlsharp -c random.sml
```

In doing this, we compile Mersenne Twister (after commenting out its `main` function).

```
$ gcc -c -o mt.o mt19937ar.c
```

We now have all the object files for the program. We can link them to an executable file by specifying the top-level interface file and the external object files referenced through `_import` declarations.

```
$ smlsharp main.smi mt.o
```

SML# analyzes `smi` file, traverse all the interface files referenced from this file, make a list of object files corresponding to the interface files, and then links all the object files with those specified in the command line argument to generate an executable file.

14.3 Structure of interface files

An interface file describes the interface of a source file to be compiled separately. Its contents consists of **Require** declarations and **Provide** declarations. Require declarations specifies the list of interface files used in this compilation unit as a sequence of declarations of the following form.

`_require` *smiFilePath*

smiFilePath is an interface file (smi file) of other compile unit. Frequently used interface files can be grouped together and given a name. SML# maintain a few of them. A useful example is `basis.smi` which contain all the (mandatory) names (types, functions etc) defined in Standard ML Basis Library. In order to make the basis library available, write the following at the beginning of its interface file.

```
_require "basis.smi";
```

Provide declarations describes the resources implemented in this compile unit. Resources corresponds to those definable as SML# declarations, including the following.

- `datatype` definitions
- `type` definitions
- `exception` declarations
- `infix` declarations
- variable definitions
- module definitions

```
fun main() =
  let
    fun getInt () =
      case TextIO.inputLine TextIO.stdIn of
        NONE => 0
      | SOME s => (case Int.fromString s of NONE => 0 | SOME i => i)
    val seed = (print "input a number of your choice (0 for exit)";
getInt())
  in
    if seed = 0 then ()
    else
      let
        val _ = Random.init seed;
        val oracle = Random.genrand()
        val message =
          " あなたの運勢は, " ^
          (case oracle mod 4 of 0 => "大吉" | 1 => "小吉"| 2 => "吉"
| 3 => "凶")
          ^ "です. \n"
        val message = print message
      in
        main ()
      end
  end
val _ = main();
```

Figure 14.1: Example of main.sml

queue.smi file：

```
_require "basis.smi"
structure Queue =
struct
  datatype 'a queue = Q of 'a list * 'a list
  exception Dequeue
  val empty : 'a queue
  val isEmpty : 'a queue -> bool
  val enqueue : 'a queue * 'a -> 'a queue
  val dequeue : 'a queue -> 'a queue * 'a
end
```

queue.sml file:

```
structure Queue =
struct
  datatype 'a queue = Q of 'a list * 'a list
  exception Dequeue
  val empty = Q ([],[])
  fun isEmpty (Q ([],[])) = true
    | isEmpty _ = false
  fun enqueue (Q(Old,New),x) = Q (Old,x::New)}
  fun dequeue (Q (hd::tl,New)) = (Q (tl,New), hd)
    | dequeue (Q ([],_) = raise Dequeue
    | dequeue (Q(Old,New) = dequeue (Q(rev New,[]))
end
```

Figure 14.2: Example of interface file

- `functor` definitions

Figure 14.2 is an example interface file `queue.smi` and its implementation `queue.sml`.

As seen in this example, Provide declarations in interface file resembles Standard ML signatures. A big difference is that in provide declarations in an interface file, `datatype` and `exception` are not specifications but they represent unique entities. `datatype 'a queue` and `exception Dequeue` declared in `queue.smi` are treated as the generative type and exception in its implementation. As a result, they corresponds to the same entity even if `queue.smi` is used in multiple interface files through `_require` declarations.

14.4 Opaque types

The principle underlying interface declaration are the following.

1. Define compile-time resources (static entities) such as `datatype` and `exception` themselves just the same way as they will be defined in a source file.

2. Declare only the types of runtime resources such as functions and variables.

They are necessary and sufficient information to compile other source file that uses this interface file. However, this principle alone cannot support ML's information hiding through opaque signatures. For example, in the previous interface file `queue.smi`, the `'a queue` type is explicitly defined to be `Q of 'a list * 'a list` and this information is open to the user of this interface file, but we often want to hide this implementation details.

To solve this problem, the interface language introduces the following opaque type declarations.

```
type tyvars tyid (= typeRep)     (* describe paranthesis as they are *)
eqtype tyvars tyid (= typeRep)     (* describe paranthesis as they are *)
```

These declaration reveals to the compiler that the internal representation of *tyid* is *typeRep* but this information is not available to the code that uses this interface file through `_require` declaration. As in signature, `type` declaration defines a type on which equality operation is not defined and `eqtype` declaration defines a type with on which equality operation is define. *typeRep* is the type constructor that implements the type *tyid*. For example, consider the following implementation:

```
type t1 = int
type t2 = int list
type 'a t3 = ('a * 'a) array
```

One can write the following in the interface file:

```
type t1 (= int)
type t2 (= list)
type 'a t3 (= array)
```

If the implementation type is either a record type, tuple type, or function type, then *typeRep* must be {}, `*`, or `->`, respectively. If the type is defined by `datatype`, then *typeRep* must be one of the following according to the definition of its constructors:

- `unit`. The type consists only of a single constructor that has no argument.

- `contag`. The type consists only of more than one constructors with no argument.

- `boxed`. Otherwise.

For example, to make `datatype 'a queue` declaration in `queue.smi`, one can write the following.

```
type 'a queue (= boxed)
```

Furthermore, as in signature, interface file need not exhaustively list all the resources the implementation may define. Only those resources defined in the interface file become visible to the code that use it through `_require` declaration.

14.5 Treatment of signatures

In Standard ML, in addition to resources explained in Section 14.3, signatures are also named resources. For example, one should be able to provide `QUEUE` signature as well as `Queue` structure. In require declaration in an interface file, signature files can be specified in the following syntax.

`_require` *sigFilePath*

sigFilePath is a path to a signature file. To understand this mechanism, let us review some properties of Standard ML signatures.

- A signature may reference some types define in some other structures.
- A signature itself does not define any type.

To deal with this situation properly, SML# compiler treats `_require` *sigFilePath* declaration as follows.

- Evaluate the signature in the file *sigFilePath* in the context generated by all the other Require declarations.
- The signature declaration is inserted at the beginning of the source file that use this interface file through `_require`.

Figure 14.3 shows an interface file containing an opaque signature.

14.6 Functor support

SML# can separately compile functor into an object file, and can be used from other compilation unit through `_require` declaration. To provide a functor, write the following in its interface file.

```
functor id(signature) =
struct
  (* the same as Provide declarations of structure *)
end
```

signature is a Standard ML signature. Below is an example of an interface file for binary search tree.

```
_require "basis.smi"
functor BalancedBinaryTree
  (A:sig
      type key
      val comp : key * key -> order
    end
  ) =
struct
  type 'a binaryTree (= boxed)
  val empty : 'a binaryTree
  val isEmpty : 'a binaryTree -> bool
  val singleton : key * 'a -> 'a binaryTree
  val insert : 'a binaryTree * A.key * 'a -> 'a binaryTree
  val delete : 'a binaryTree * key -> 'a binaryTree
  val find : 'a binaryTree * A.key -> 'a option
end
```

In using functors in separate compilation, one should note the following.

- **Functor is not a mechanism for separate compilation.** In some existing practice of ML, probably due to the lack of separate compilation, functors can be used to compile some modules independently from the others. For example, if one write

queue-sig.sml file:

```
signature Queue =
sig
  datatype 'a queue = Q of 'a list * 'a list
  exception Dequeue
  val empty : 'a queue
  val isEmpty : 'a queue -> bool
  val enqueue : 'a queue * 'a -> 'a queue
  val dequeue : 'a queue -> 'a queue * 'a
end
```

queue.smi file:

```
_require "basis.smi"
_require "queue-sig.sml"

structure Queue =
struct
  type 'a queue (= boxed)
  exception Dequeue
  val empty : 'a queue
  val isEmpty : 'a queue -> bool
  val enqueue : 'a queue * 'a -> 'a queue
  val dequeue : 'a queue -> 'a queue * 'a
end
```

queue.sml file:

```
structure Queue : QUEUE =
struct
  datatype 'a queue = Q of 'a list * 'a list
  exception Dequeue
  val empty = Q ([],[])
  fun isEmpty (Q ([],[])) = true
    | isEmpty _ = false
  fun enqueue (Q(Old,New),x) = Q (Old,x::New)}
  fun dequeue (Q (hd::tl,New)) = (Q (tl,New), hd)
    | dequeue (Q ([],_) = raise Dequeue
    | dequeue (Q(Old,New) = dequeue (Q(rev New,[]))
end
```

Figure 14.3: Example of interface file with signatures

A.sml file:

```
structure A =
struct
  ...
end
```

B.sml file:

```
structure B =
struct
  open A
  ...
end
```

then `B.sml` file directly depends on `A.sml` file. If one rewrite `B.sml` using a functor as below then this dependency can be avoided.

B.sml file:

```
functor B(A:sig ... end) =
struct
  open A
  ...
end
```

This is exactly what separate compilation achieves. A system such as SML# where a complete separate compilation is supported, this form of functor usage is unnecessary and undesirable.

- **Usage of functor incurs some overhead.** Functors can take types as parameters and therefore strictly more powerful than polymorphic functions. However, this type parameterization requires the compiler to generate code that behaves differently depending on the argument types. The resulting code inevitably incurs more overhead than the corresponding code with the type argument predetermined (i.e. ordinary structures). The programmer who use functor should be aware of this const and restrict functors in cases where the advanced feature of explicit type parameterization is really required.

In the current version of SML# has the following limitation on the usage of functors.

- If a functor has a formal abstract type constructor with type arguments, only heap-allocated internal representations (such as `array`, `boxed`, `{}`, `->`, and `*`) can be applied to the formal type constructor. The following example causes a compile error in SML#.

```
# functor F(type 'a t) = struct end structure X = F(type 'a t = int);
(interactive):2.17-2.34 Error:
(name evaluation "440") Functor parameter restriction: t
```

14.7 Replications

An interface file describes resources themselves. So, for example, for a module of the form

```
structure Foo =
struct
  structure A = Bar
  structure B = Bar
...
end
```

one cannot write its interface as

```
structure Foo =
struct
  structure A : SigBar
  structure B : SigBar
```

```
...
end
```

The interface language mechanism so far describes forces us to repeat most of the contents of `Bar` twice. To suppress this redundancy, the interface language allow replication declaration of the following form.

- `structure` *id* `=` *path*
- `exception` *id* `=` *path*
- `datatype` *id* `= datatype` *path*
- `val` *id* `=` *path*

If you know that in advance that two resources are replication of the same resources, then you can declare them as replication using this mechanism in an interface file. For example, `Bar` structure is provided by an interface file `bar.smi` and that you know that both `A` and `B` should be replications of `Bar`, then you can simply write the following interface file.

```
_require "bar.smi"
structure Foo =
struct
  structure A = Bar
  structure B = Bar
...
end
```

14.8 Top-level execution

Execution of an SML# program is done by evaluating the top-level declarations. If a program consists of a single file then this model is realized by generating the code for each declaration and concatenate them in the order of declarations. However, a separately compiled program consisting of multiple files, we have to decide the order of execution of declarations. SML# decides the order based on the following policy.

1. The top-level declarations corresponding to the top-level `.smi` file, i.e. the file give as a command line parameter to `smlsharp` command, are executed after the declarations corresponding to any other `.smi` files. In what follows, we refer to this top-level as link top-level.
2. If a variable provided by `A.sml` is used in `B.sml`, the top-level of `A.sml` is executed before executing that of `B.sml`.
3. If `A.sml` is not reachable from the link top-level through the provide-use relationship, the top-level of `A.sml` is not executed (the compiler does not link `A.o` into the executable file).

Execution order of declarations respects the dependency of references of names. Note that the reference relation is the actual relation generated by occurrences of long-ids in a program, and not the relation generated by `_require` declarations in `.smi` files.

There are two exceptions. One is the link top-level, whose top-level is always executed. Another is the require declaration of the following form:

`_require` *smiFilePath* `init`

This forces the compiler to link the object file corresponding to *smiFilePath* and generate code executing its toplevel. This form is only for special purpose; it had better not to abuse it.

Part III

Reference manual

Chapter 15

Introduction

This part defines the specification of SML# system, including the language, the library, the `smlsharp` command, and its external interfaces. As we stated in Part I, SML# maintain the backward compatibility to Standard ML. In addition to the definitions of the language syntax, the Definition of Standard ML [5] defines the static and dynamic semantics of the language as formal derivation systems of semantic judgments. Different from Standard ML as stated in the Definition, which is a self-contained "closed system", SML# is an open system that include direct C language interfaces and seamless integration of SQL. At this moment, we do not have an appropriate means to formally define the semantics of such an open system. So in this reference manual, we do not provide a formal definition of the static and dynamic semantics; instead, we explain, in English, typing properties and the meanings of language constructs whenever appropriate.

15.1 Notations

In the definitions, the following notations are used.

- Terminal symbols are written as `"SML#"` using a typewriter font.
- Non-terminal symbols are written as $\langle exp \rangle$ with angle brackets.
- (opt)? represents that opt is optional.
- For a syntax class $\langle x \rangle$, the notation $\langle xList \rangle$ represents list of one of more $\langle x \rangle$'s separated by white space, and $\langle xSeq \rangle$ denotes one of the following.

$\langle xSeq \rangle$	::=	$\langle x \rangle$	(one element)
	|		(empty)
	|	`(` $\langle x \rangle_1$ `,` $\cdots$ `,` $\langle x \rangle_n$ `)`	(n-ary tuple)

Chapter 16

The SML# Structure

This chapter explains the language structure of SML#, the typing concept, and its evaluation model.

16.1 Programs in the interactive mode

An interactive mode program of SML# is a sequence of declarations, terminated by semicolon (`;`). The following shows a simple interactive program session of SML#

```
$ smlsharp
SML# 4.2.0 ...
# fun fact 0 = 1
>   | fact n = n * fact (n - 1);
> val x = fact 10;
val fact = fn : int -> int
val x = 3628800 : int
```

`#` and `>` are the first-line prompt character and the second-line prompt character, respectively. As shown in this example, the SML# interactive compiler print the result of evaluation of the input program.

Declarations are divided into the core language declarations ⟨*decl*⟩, which generates values such as functions and records, and the module language declarations ⟨*strDecl*⟩, which are named collections of core language declarations.

⟨*interactiveProgram*⟩ ::= `;`
| ⟨*decl*⟩ ⟨*interactiveProgram*⟩
| ⟨*topdecl*⟩ ⟨*interactiveProgram*⟩

- Core language declarations

  The following shows core language declarations (⟨*decl*⟩) and simple examples.

⟨*decl*⟩	::=	⟨*infixDecl*⟩	infix declaration
	|	⟨*valDecl*⟩	val delaration
	|	⟨*valRecDecl*⟩	val rec delaration
	|	⟨*funDecl*⟩	function delaration
	|	⟨*datatypeDecl*⟩	data type delaration
	|	⟨*typeDecl*⟩	type alias delaration
	|	⟨*exceptionDecl*⟩	exception delaration
	|	⟨*localDecl*⟩	local declaration

Declaration kind	Simple example
infix declaration	`infix 4 =`
val declaration	`val x = 1`
val rec declaration	`val rec f = fn x => if x = 0 then 1 else x * f (x - 1)`
function declaration	`fun f x = if x = 0 then 1 else x * f (x - 1)`
datatye declaration	`datatype foo = A | B`
type declaration	`type person = {name:string, age:int}`
exception declaration	`exception Fail of string`
local declaration	`local val x = 2 in val y = x + x end`

- Module language declarations

  The following shows module language declarations (⟨*strDecl*⟩) and simple examples.

⟨*topdecl*⟩	::=	⟨*strDecl*⟩	structure delaration
	|	⟨*sigDecl*⟩	signature delaration
	|	⟨*functorDecl*⟩	functor delaration
	|	⟨*localTopdecl*⟩	local declaration

structure declaration	`structure Version =`
	`  struct`
	`  val version = "2.0.0"`
	`  end`

signature declaration	`signature VERSION =`
	`  sig`
	`  val version :string`
	`  end`

functor declaration	`functor System(V:VERSION) =`
	`  struct`
	`  val name = "SML#"`
	`  val version = V.version`
	`  end`

local declaration	`local`
	`  structure V = Version`
	`in`
	`  structure Release = struct`
	`  val version = V.version`
	`  val date = "2025-03-24"`
	`  end`
	`end`

16.1.1 Evaluation of core language declarations

Execution of an interactive program is done by evaluating a list of declaration sequentially. Evaluation of a core language declaration has the effect of evaluating the components, such as expressions, contained in the declaration, and binding the identifiers defined in the declaration to the values generated by the evaluation of the components. Generated values are either static or dynamic.

Static values are compile-time value generated by the compiler. They are one of the following.

infix operator status This indicates that an identifier is parsed as an infix operator. It also has the associatibity (right associative or left associative) and association strength (from 0 to 9).

constructor status This indicates that an identifier status is a data constructor. An identifier with the constructor status only matches with the corresponding constructor in pattern marching.

type constructor Type constructors are newly defined types generated by datatype declarations. Parametric types with type parameters can be defined.

type Types specify a static property of a dynamic values produced by executing the corresponding program fragment.

The SML# compiler statically evaluates declarations and generates static values, and bind names defined in the declarations to those static values. For declarations such as ⟨*valDecl*⟩ that implies program execution, the compiler generates executable code, which when executed, constructs dynamic values at runtime and binds names to those dynamic values.

Dynamic values are one of the following.

Built-in values Runtime data of built-in types defined in Chapter 18. They include atomic values, lists, arrays, and vectors. For example, values of type `int32` are 32-bit signed integers whose representation is defined by the underlying machine architecture.

Function closures They are runtime representations of values of function types.

Data constructors They are built-in functions to construct datatype representations.

Exception closures They are built-in functions to construct exception values.

Records They are values of record and tuple types.

Datatype representation They are generated by data constructors defined in datatype declarations.

Exception values They are generated by exception constructors.

The following show a summary of the generated static and dynamic values for each of declarations.

declaration class	static values bindings	dynamic values bindings	assigned to
infix declaration	infix status		variables, constructor n
val delaration	types	dynamic values corresponding to types	variables
val rec delaration	types	function closures	variables
function delaration	types	function closures	variables
data type delaration	type constructors constructor status	 data constructors	type constructor names data constructor names
type alias delaration	type function		type constructor names
exception delaration	constructor status	exception constructors	exception constructor n
exception delaration	depending on the contents	depending on the contents	depending on the conte

We outline declaration evaluation and the resulting value bindings below. The detailed syntax and semantics of each of these declaration classes are given in Chapter 23.

⟨*infixDecl*⟩ This is a declaration of the form `infix` *id*. The identifier *id* is given infix operator property. *n* is optional. The following shows a simple example.

```
# infix 7 *;
# infix 8 ^;
# fun x ^ y = if y = 0 then 1 else x * x ^ (y - 1);
val ^ = fn : int * int -> int
# val a = 4 * 3 ^ 2
val a = 36 : int
```

Since `*` and `^` has infix precedence 7 and 8, respectively, `4 * 3 ^ 2` is elaborated to `*(4, ^ (3,2))`.

⟨*valDecl*⟩ This is a declaration of the form `val` *pat* `=` *exp*. Evaluation is done by evaluating *exp* and checking whether the resulting dynamic value matches *pat*. If it matches then variables in *pat* are bound to the corresponding values of the result. The simplest example is the following variable binding.

```
# val x = 1;
val x = 1 : int
```

In this example, variable `x` is bound to the result of evaluating expression `1`, namely static type `int` and dynamic value 1.

```
# val (x, y) = (1, 2);
val x = 1 : int
val y = 2 : int
```

The details of expressions ⟨*exp*⟩ and patterns are given in Chapter 19 and 20. .

⟨*valRecDecl*⟩ Val declarations restricted to (mutually recursive) functions.

```
# val rec even = fn x => if x = 0 then true else odd (x - 1)
> and odd = fn x => if x = 1 then true else odd (x - 1);
val even = fn : int -> bool
val odd = fn : int -> bool
```

In this declaration, each identifier is bound to the corresponding type and function value.

⟨*funDecl*⟩ This is for mutually recursive function definitions.

```
# fun even x = if x = 0 then true else odd (x - 1)
> and odd x = if x = 1 then true else odd (x - 1);
val even = fn : int -> bool
val odd = fn : int -> bool
```

In this declaration, each identifier is bound to the corresponding type and function value.

⟨*datatypeDecl*⟩ This defines new mutually recursive type constructors.

```
# datatype foo = A of int | B of bar and bar = C of bool | D of foo;
datatype bar = C of bool | D of foo
datatype foo = A of int | B of bar
# D (A 3);
val it = D (A 3) : bar
```

Evaluation of this declaration generates two new type constructors (with no type parameter) `foo` and `bar` and the identifiers `foo` and `bar` are bound to them. It also generate data constructors `A`, `B` and `C`, `D` for `foo` and `bar`, respectively, and the identifiers to them. These identifiers are given the constructor status.

As in the above explanation, in this manual, we generally identify type constructors such as `foo` and data constructors such as `A` with their names `foo` and `A`.

⟨*typeDecl*⟩ This bind an identifier to a type or a type function.

```
# type 'a foo = 'a * 'a;
type 'a foo = 'a * 'a
# fun f (x:int foo) = x;
val f = fn : int * int -> int * int
```

Evaluation of this generates a type function that takes a type parameter represented by `'a` and returns a type `'a * 'a`, and binds identifier `foo` to this type function. In the scope of this declaration, τ `foo` is used as an alias of τ `*` τ.

⟨*exceptionDecl*⟩ This defines exception constructors.

```
# exception Foo of int;
exception Foo of int
```

Evaluation of this declaration generates a new exception constructor with a parameter of type `int`. In the scope of this declaration, the identifier `foo` is given the constructor status and is bound to the exception constructor.

⟨*localDecl*⟩ Declarations between `local` and `in` are local until `end`.

```
# local
>   val x = 2
> in
>   val y = x + x
> end;
val y = 4 : int
```

The scope of the declaration `val x = 2` is until `end`. The variable `x` is not visible from outside of this local declaration and therefore only `y` is printed in the interactive session.

16.1.2 Evaluation of module language declarations

A structure declaration, the main component of the module language, is a mechanism to bundle a list of declarations and gives it a name. Evaluation of a structure expression yields a static type environment representing the static binding of the declarations, and a dynamic value environment representing the dynamic binding of the declarations. The effect of evaluation of a structure declaration is to extend the current type environment and the current value environment with the static and dynamic environment obtained from the generated environments by prefixing the bound names in the environments with the structure name.

For example, the structure expression

```
struct
  val version = "4.2.0"
end
```

generates the type environment {`version` : `string`} and the runtime environment {`version` : `"3.4.0"`}. Therefore the structure declaration

```
structure Version =
  struct
    val version = "4.2.0"
  end
```

has the effect of extending the current type environment and the current runtime-environment with the following binding of long names: {`Version.version` : `string`}, and {`Version.version` : `"3.4.0"`}.

A signature constraint can be specified to a structure expression. A signature statically constrains the type environment generated by the structure. In addition to type constraints to variables, constraints of type declarations can be specified. A structure expression with a signature constraint generates a type environment that only contains those names that are specified in the signature.

A functor is a function that takes a structure and returns a structure. Different from functions in the core language functor definitions are restricted to first-order and top-level.

We show evaluation of module language declarations in the interactive mode below. The detailed syntax and semantics of each of these declaration classes are given in Chapter 24.

⟨*strDecl*⟩ This declaration defines a structure.

```
# structure Version =
>   struct
>     val version = "4.2.0"
>   end;
structure Version =
struct
  val version = "4.2.0" : string
end
```

Evaluation of this declaration binds `Version` to a type environment representing the structure containing `version` component, and long identifier `Version.version` to `string` type and the dynamic value `"4.2.0"`.

⟨*sigDecl*⟩ This declaration binds an identifier to a signature.

```
# signature VERSION =
>  sig
>    val version : string
>  end;
signature VERSION =
struct
  val version : string
end
```

Evaluation of this declaration binds `VERSION` to the specified signature. Signatures describes types of components of SML# structures.

⟨*functorDecl*⟩ This declaration defines a functor, which is a function taking a structure and returning a structure.

```
functor System(V:VERSION) =
  struct
    val name = "SML#"
    val version = V.version
  end;
```

Evaluation of this declaration binds the identifier `System` to a functor that takes a structure of `VERSION` signature and returns a structure containing `name` and `version` components.

⟨*localTopdecl*⟩ Declarations between `local` and `in` are local until `end`.

```
# local
>   structure V = Version
> in
>   structure Release = struct
>     val date = "2025-03-24"
>     val version = V.version
>   end
> end;
val Release =
struct
  val date = "2025-03-24" : string
  val version = "4.2.0" : string
end
```

The scope of the declaration `structure V = Version` is until `end`. The structure `V` is not visible from outside of this local declaration and therefore only `Release` is printed in the interactive session.

16.2 Programs in the separate compilation mode

A program in the separate compilation mode is a set of source files and their interface files. The contents of a source file, denoted here as ⟨*source*⟩, are given below.

⟨*source*⟩ ::= (⟨*interfaceFileSpec*⟩)? ⟨*sourceProgram*⟩
⟨*interfaceFileSpec*⟩ ::= _interface " ⟨*filePath*⟩ "
⟨*sourceProgram*⟩ ::=
| ⟨*decl*⟩ ⟨*sourceProgram*⟩
| ⟨*topdecl*⟩ ⟨*sourceProgram*⟩

If there is no interface file specification ⟨*interfaceFileSpec*⟩ in a source file and there is a file of the same name *S*`.smi` as that of the source file *S*`.sml` (except for its suffix) in the same directory, then that file is implicitly specified as the interface file of the source file. ⟨*filePath*⟩ is a relative path. Regardless of OS, `/` (slash) is used for directory separator.

The contents of an interface file, ⟨*interface*⟩, consist of ⟨*requireList*⟩ that specifies the set of interface file paths required for the source file, and ⟨*provideList*⟩ that specifies the set of declarations the source file provides to the other compilation units.

⟨*interface*⟩ ::= ⟨*requireList*⟩ ⟨*provideList*⟩

⟨*requireList*⟩ ::=
| `_require` (`local`)? ⟨*interfaceName*⟩ ⟨*requireList*⟩ (`init`)?
| `_require` (`local`)? ⟨*sigFilePath*⟩ ⟨*requireList*⟩ (`init`)?

⟨*interfaceName*⟩ ::= ⟨*smiFilePath*⟩
| ⟨*librarySmiFilePath*⟩

⟨*provideList*⟩ ::=
| ⟨*provide*⟩ ⟨*provideList*⟩

⟨*provide*⟩ ::= ⟨*provideInfix*⟩
| ⟨*provideVal*⟩
| ⟨*provideType*⟩
| ⟨*provideDatatype*⟩
| ⟨*provideException*⟩
| ⟨*provideStr*⟩
| ⟨*provideFun*⟩

⟨*requireList*⟩ specifies interface file names (file path or library name) of source files or signature file paths referenced by the source file. `local` directive in a `_require` declaration indicates that the specified interface file is only used inside of the source file and does not referenced in ⟨*provideList*⟩. `init` is an annotation indicating that the program corresponding to the specified interface must be executed and precede the program that `_require`s it. ⟨*provideList*⟩ specifies the set of declarations the source file provides to the other compilation units.

⟨*smiFilePath*⟩, ⟨*sigFilePath*⟩, and ⟨*librarySmiFilePath*⟩ are file paths relative to the file in which `_require` is written. Regardless of OS, `/` (slash) is used for directory separators. If a path does not begins with `.` (period) and the specified file does not exist, the compiler searches for the file from the load path given in the command line.

SML# provides the following interface libraries.

library smi file name	contents
basis.smi	Standard ML Basis Library
ml-yacc-lib.smi	yacc and lex tools
smlformat-lib.smi	SMLFormat formatter generator
smlnj-lib.smi	Standard ML of New Jersey library
ffi.smi	C FFI support library
thread.smi	Multithread library
reify.smi	Dynamic typing library
smlunit-lib.smi	SMLUnit tool

The following summarizes ⟨*provide*⟩ specifications corresponding to declarations.

- Correspondence between core declarations and their interfaces

declaration (⟨*decl*⟩)	corresponding provide (⟨*provide*⟩)
⟨*infixDecl*⟩	⟨*provideInfix*⟩
⟨*valDecl*⟩	⟨*provideVal*⟩
⟨*valRecDecl*⟩	⟨*provideVal*⟩
⟨*funDecl*⟩	⟨*provideVal*⟩
⟨*datatypeDecl*⟩	⟨*provideData*⟩
⟨*typeDecl*⟩	⟨*provideType*⟩
⟨*exceptionDecl*⟩	⟨*provideException*⟩
⟨*localDecl*⟩	

- Correspondence between module delcarations and their interfaces

declaration (⟨*topdecl*⟩)	corresponding provide (⟨*provide*⟩)
⟨*strDecl*⟩	⟨*provideStr*⟩
⟨*sigDecl*⟩	
⟨*functorDecl*⟩	⟨*provideFun*⟩
⟨*localTopdecl*⟩	

Signature files are directly referenced in the interface through `_require` ⟨*sigFilePath*⟩ .

We show simple examples below.

- Example 1 (Using the Standard ML Basis Library)

file	code
`hello.sml`	`val _ = print "Welcome to SML #\n"`
`hello.smi`	`_require "basis.smi"`

Compilation and execution

```
$ smlsharp hello.sml
$ ./a.out
Welcome to SML #$
```

- Example 3 (separate compilation)

file	code
`hello.sml`	`val _ = puts "Welcome to SML#"`
`hello.smi`	`_require "puts.smi"`
`puts.sml`	`val puts = _import "puts" : string -> int`
`puts.smi`	`val puts : string -> int`

Compilation and execution

```
$ smlsharp -c hello.sml
$ smlsharp -c puts.sml
$ smlsharp -o hello hello.smi
$ ./hello
Welcome to SML# $
```

16.3 Major Components of SML# Programs

Before defining the syntax and semantics of these declarations and interfaces, we define those of the following elements that construct these declarations and interfaces.

Lexical structures The definition of keywords and identifiers (Chapter 17).

Expressions The major components of programs made from lexical components. They have static types and generates dynamic values (Chapter 19).

Types Used for type annotations of expressions and identifiers and the definition of interfaces (Chapter 18).

Patterns Denoting the structure of values received by functions and value bindings. (Chapter 20).

Scoping rules The definition of scopes of identifiers and dynamic value bindings. (Chapter 21).

SQL expressions The expressions denoting SQL commmands that is seamlessly integrated into SML#. Similarly to other expressions, they have polymorphic types and can be used with other expressions. (Chapter 22).

Then, we define the syntax and semnatics of core declarations in Chapter 23 and those of module declarations in Chapter 24.

Chapter 17

Lexical structure

This chapter defines the set of characters and the lexical structures used in SML#. Definitions of lexical structures use standard notations for regular expressions.

17.1 Character set

Characters usable in the SML# language are extended ASCII characters from 0 to 255 (in decimal). Key words and delimiters are among the characters from 0 through 127. Identifiers may ASCII characters from 0 to 255, including 8-bit characters from 128 to 255, as far as they are not overlap with keywords and delimiters. This rule allows the programmer to use `UTF8` encoded Japanese (and other) character string as identifiers. The following interactive session show example codes with Japanese identifiers.

```
$ smlsharp
SML# 4.2.0 ...
# datatype メンバー =
>   研究員 of {氏名:string, 年齢:int, 学位:string}
> | 職員 of {氏名:string, 年齢:int};
datatype メンバー =
    研究員 of {学位:string, 年齢:int, 氏名:string}
  | 職員 of {年齢:int, 氏名:string}
# 研究員 {学位="Ph.D.", 年齢=21, 氏名="大堀"};
val it =
  研究員 {学位 = "Ph.D.", 年齢 = 21, 氏名 = "大堀"} : メンバー
```

17.2 Lexical items

SML# lexical items consist of the following.

keywords The following are SML# keywords and they cannot be used as identifiers.

```
abstype and andalso as case datatype do else end eqtype exception fn fun functor
handle if in include infix infixr let local nonfix of op open orelse raise
rec set sharing sig signature struct structure then type val where while with
withtype ( ) [ ] { } , : :> ; ... _ = => -> #
```

⟨*alphaId*⟩ defined in the identifier section does not contain these keywords.

SQL keywords The following are keywords used in SQL expressions and cannot be used as identifiers within SQL expressions.

```
asc all begin by commit cross default delete desc distinct fetch first from
group inner insert into is join limit natural next not null offset only on
or order rollback row rows select set update values where
```

These strings are not among the ⟨*alphaId*⟩ in the SQL expression that begins with the `_sql` keyword introduced in Section 19 and defined in Chapter 22. SQL expressions are not expressions in the definition of Standard ML, this restriction preserves the backward compatibility.

Extended keywords The following names started with `_` are keywords used to represent SML# special features. Since they are not lexical items in the definition of Standard ML, introducing them preserves the backward compatibility.

```
__attribute__ _builtin _foreach _import _interface _join _dynamic _dynamiccase
_polyrec _require _sizeof _sql _sqlserver _typeof _use
```

Identifiers Identifiers are names used in programs. SML# has the following 7 classes of identifiers defined below. Their classes are determined from their occurring context, so the same name can be used in different identifiers.

name	usage
⟨*vid*⟩	variables, data constructors
⟨*lab*⟩	record labels
⟨*strid*⟩	structure names
⟨*sigid*⟩	signature names
⟨*funid*⟩	functor names
⟨*tycon*⟩	type constructors
⟨*tyvar*⟩	type variables

これら識別子の構造は以下の通りである.

⟨*vid*⟩ ::= ⟨*alphaId*⟩ | ⟨*symbolId*⟩
⟨*lab*⟩ ::= ⟨*alphaId*⟩ | ⟨*string*⟩ | ⟨*decimal*⟩ | ⟨*decimal*⟩ `_` ⟨*alphaId*⟩ (note *1)
⟨*strid*⟩ ::= ⟨*alphaId*⟩
⟨*sigid*⟩ ::= ⟨*alphaId*⟩
⟨*funid*⟩ ::= ⟨*alphaId*⟩
⟨*tyvar*⟩ ::= `'` ⟨*alphaId*⟩ | `''` ⟨*alphaId*⟩
⟨*tycon*⟩ ::= ⟨*alphaId*⟩ | ⟨*symbolId*⟩

Note.

1. There are the following three kinds of record labels: character string labels (⟨*alphaId*⟩ or ⟨*string*⟩), integer labels(⟨*decimal*⟩), ordered character string labels(`[1-9]` `[0-9]`* `_` ⟨*alphaId*⟩). In SML#, record fields are sorted according to the ordering of their labels. Character string labels are ordered by `String.compare`. Integer labels are ordered according to the integer they represents. Ordered character string labels are order by the lexicographical pairing of integer labels and character string labels.

The definition of these character classes are given below.

⟨*alpha*⟩ ::= [A-Za-z\127-\255]
⟨*symbol*⟩ ::= `!` | `%` | `&` | `$` | `+` | `/` | `:` | `<` | `=` | `>` | `?` | `@` | `~` | `` ` `` | `|` | `#` | `-` | `^` | `\`
⟨*alphaId*⟩ ::= ⟨*alpha*⟩ (⟨*alpha*⟩ | `[0-9]` | `'` | `_`)* (Note 1)
⟨*decimal*⟩ ::= `[1-9]` `[0-9]`*
⟨*symbolId*⟩ ::= ⟨*symbol*⟩ * (Note 2)

Note.

1. The character class ⟨*alphaId*⟩ does not contain keywords. Furthermore, in SQL expressions that begins with `_sql`, ⟨*alphaId*⟩ does not contain SQL keywords.
2. ⟨*symbolId*⟩ does not contain keywords. Therefore `==>` is an instance of ⟨*symbolId*⟩ but `=>` is not.

Long identifiers For ⟨*vid*⟩ (variable names and constructor names) and ⟨*strid*⟩ (structure names), the following long identifiers are defined.

⟨*longVid*⟩	::=	(⟨*strid*⟩ `.`)* ⟨*vid*⟩
⟨*longTycon*⟩	::=	(⟨*strid*⟩ `.`)* ⟨*tycon*⟩
⟨*longStrid*⟩	::=	(⟨*strid*⟩ `.`)* ⟨*strid*⟩

constant literals ⟨*scon*⟩ Syntax for constant literals are given below.

⟨*scon*⟩	::=	⟨*int*⟩ | ⟨*word*⟩ | ⟨*real*⟩ | ⟨*string*⟩ | ⟨*char*⟩	Constant literals
⟨*int*⟩	::=	(`~`)?[`0-9`]+	Decimal integers
	|	(`~`)?`0x`[`0-9a-fA-F`]+	Hexadecimal integers
⟨*word*⟩	::=	`0w`[`0-9`]+	unsigned decimal integers
	|	`0wx`[`0-9a-fA-F`]+	unsigned hexadecimal integers
⟨*real*⟩	::=	(`~`)?[`0-9`]+ `.` [`0-9`]+ [`Ee`](`~`)?[`0-9`]+	Floating-point numbers
	|	(`~`)?[`0-9`]+ `.` [`0-9`]+	
	|	(`~`)?[`0-9`]+ [`Ee`](`~`)?[`0-9`]+	
⟨*char*⟩	::=	`#"` (⟨*printable*⟩ | ⟨*escape*⟩) `"`	Character
⟨*string*⟩	::=	`"` (⟨*printable*⟩ | ⟨*escape*⟩)∗ `"`	String
⟨*printable*⟩	::=	characters except for `\` and `"`	
⟨*escape*⟩	::=	`\a`	The warning character (ASCII 7)
	|	`\b`	Backspace (ASCII 8)
	|	`\t`	Horizontal tab (ASCII 9)
	|	`\n`	New line character (ASCII 10)
	|	`\v`	Vertical tab (ASCII 11)
	|	`\f`	Home feed (ASCII 12)
	|	`\r`	Carriage return (ASCII 13)
	|	`\^`[\064-\095]	Control character represented by [\064-\095]
	|	`\\`	The \ character
	|	`\"`	The " character
	|	*ddd*	The character of decimal code *ddd*
	|	\$f \cdots f$\	ignoring whitespace characters $f \cdots f$
	|	`\u`*xxxx*	The string of the UTF-8 character of hexadecima

Chapter 18

Types

This chapter defines the syntax for types and describes the built-in types.

Types are divided into monotypes (⟨*ty*⟩) and polytypes(⟨*polyTy*⟩). The syntax for mono types are given below.

⟨*ty*⟩	::=	⟨*tyvar*⟩	type variable names
	|	`{` (⟨*tyrow*⟩)? `}`	record types
	|	⟨ty_1⟩ `*` $\cdots$ `*` ⟨ty_n⟩	tuple types ($n \geq 2$)
	|	⟨*ty*⟩ `->` ⟨*ty*⟩	function types
	|	(⟨*tySeq*⟩)? ⟨*longTycon*⟩	(parameterlized) datatypes
	|	`(` ⟨*ty*⟩ `)`	
⟨*tyrow*⟩	::=	⟨*lab*⟩ `:` ⟨*ty*⟩ (`,` ⟨*tyrow*⟩)?	record field types

⟨*tyvar*⟩ are type variable names. As defined in Section 17.2, they are written as `'a`, `'foo` or `''a`, `''foo`. The latter form are for those equality type variables, which range only over types that admit equality. An type admits equality, called eqtype, is any type that does not contain function type constructor and built-in types that does not admit equality. A user defined datatype is an eqtype if it only contains eqtypes. For example, τ `list` defined below is an eqtype if τ is an eqtype.

```
datatype 'a list = nil | :: of 'a * 'a list
```

The function type constructor `->` associates to the right so that `int -> int -> int` is interpreted as `int -> (int -> int)`.

⟨*longTycon*⟩ is a type constructor names defined by `datatype` declarations. Atomic types such as `int` are type constructors without type parameters (⟨*tySeq*⟩). SML#4.2.0 supports the following built-in atomic types and type constructors.

type constructor name	description	eqtype?
`int`	32 bit long signed integers	Yes
`int64`	64 bit long signed integers	Yes
`int16`	16 bit long signed integers	Yes
`int8`	8 bit long signed integers	Yes
`intInf`	unbounded signed integers	Yes
`word`	32 bit long unsigned integers	Yes
`word64`	64 bit long unsigned integers	Yes
`word16`	8 bit long unsigned integers	Yes
`word8`	8 bit long unsigned integers	Yes
`real`	floating point numbers)	No
`real32`	32 floating point numbers	No
`char`	characters	Yes
`string`	strings	Yes
`exn`	exceptions	No
`unit`	unit values (`()`)	Yes
τ `ref`	references (pointers)	Yes
τ `array`	arrays	Yes
τ `vector`	vectors	τ が eq 型ならば Yes

The `unit` type is distinguished from the empty record type `{}` as a different type. This is the modification that is not backward-compatible to Standard ML.

The syntax for polytypes (⟨*polyTy*⟩) are given below.

⟨*polyTy*⟩	::=	⟨*ty*⟩
	|	`[` ⟨*boundtyvarList*⟩ `.` ⟨*ty*⟩ `]`
	|	⟨*ty*⟩ `->` ⟨*polyTy*⟩
	|	⟨*polyTy*⟩ `*` ⋯ `*` ⟨*polyTy*⟩
	|	`{` (⟨*polyTyrow*⟩)? `}`
⟨*boundtyvarList*⟩	::=	⟨*boundtyvar*⟩ (`,` ⟨*boundtyvarList*⟩)?
⟨*boundtyvar*⟩	::=	⟨*tyvar*⟩ ⟨*kind*⟩
⟨*kind*⟩	::=	
	|	`#{` ⟨*tyrow*⟩ `}`
	|	`#` ⟨*kindName*⟩ ⟨*kind*⟩
⟨*kindName*⟩	::=	`boxed` | `unboxed` | `reify` | `eq`
⟨*polyTyrow*⟩	::=	⟨*lab*⟩ `:` ⟨*polyTy*⟩ (`,` ⟨*polyTyrow*⟩)?

- `[` ⟨*boundtyvarList*⟩ `.` ⟨*ty*⟩ `]` is the polymorphic type that makes the scope of bounded type variables ⟨*boundtyvarList*⟩ explicitly.
- A bound type variable may have the following kind constraints ⟨*kind*⟩ . The record kind `#{` ⟨*tyrow*⟩ `}` restricts the range of the bound type variable to record types that have at least fields indicated by ⟨*tyrow*⟩ . The `boxed` kind restricts it to the type of heap-allocated values. The `unboxed` kind is the complement of the `boxed` kind. The `eq` kind restricts it to eqtypes. The `reify` kind is just an annotation that the type reification feature is required and therefore does not restrict the range of the bound type variable.

The set of polytypes is the extension of the set of polytypes in the definition of Standard ML with record polymorphism, overloading and rank-1 polymorphism.

The following examples use a rank-1 polytype.

```
# fn x => (fn y => (x,y), nil);
val it = fn : ['a. 'a -> ['b. 'b -> 'a * 'b] * ['b. 'b list]]
```

In the interactive session, in addition to the above kinds, you may see the overload kind `::` `{` ⟨*tyList*⟩ `}`, which restricts the range of the bound type variable to ⟨*tyList*⟩ . The current system restricts overloading to system defined primitives, and type variables with overload kind are not allowed in a user program.

Chapter 19

Expressions

The syntax for expressions (⟨*exp*⟩) is hierarchically defined below using infix operator expressions (⟨*infix*⟩), function application expressions (⟨*appexp*⟩), atomic expressions (⟨*atexp*⟩).

- expressions (top-level)

⟨*exp*⟩	::=	⟨*infix*⟩	
	|	⟨*exp*⟩ `:` ⟨*ty*⟩	
	|	⟨*exp*⟩ `andalso` ⟨*exp*⟩	
	|	⟨*exp*⟩ `orelse` ⟨*exp*⟩	
	|	⟨*exp*⟩ `handle` ⟨*match*⟩	
	|	`raise` ⟨*exp*⟩	
	|	`if` ⟨*exp*⟩ `then` ⟨*exp*⟩ `else` ⟨*exp*⟩	
	|	`while` ⟨*exp*⟩ `do` ⟨*exp*⟩	
	|	`case` ⟨*exp*⟩ `of` ⟨*match*⟩	
	|	`fn` ⟨*match*⟩	
	|	`_import` ⟨*string*⟩ `:` ⟨*cfunty*⟩	importing C function
	|	⟨*exp*⟩ `:` `_import` ⟨*cfunty*⟩	importing C function
	|	`_sizeof(` ⟨*ty*⟩ `)`	size of type
	|	`_dynamic` ⟨*exp*⟩ `as` ⟨*ty*⟩	Dynamic type cast
	|	`_dynamiccase` ⟨*exp*⟩ `of` ⟨*match*⟩	case branches with dynamic type cast
	|	`_sqlserver` (⟨*appexp*⟩)? `:` ⟨*ty*⟩	SQL servers
	|	`_sql` ⟨*pat*⟩ `=>` ⟨*sqlfn*⟩	SQL execution function
	|	`_sql` ⟨*sql*⟩	SQL query fragments

⟨*match*⟩ ::= ⟨*pat*⟩ `=>` ⟨*exp*⟩ (`|` ⟨*match*⟩)? pattern matching

- infix operator expressions

⟨*infix*⟩	::=	⟨*appexp*⟩	
	|	⟨*infix*⟩ ⟨*vid*⟩ ⟨*infix*⟩	

- function application expressions

⟨*appexp*⟩	::=	⟨*atexp*⟩	
	|	⟨*appexp*⟩ ⟨*atexp*⟩	left associative function applications
	|	⟨*appexp*⟩ `#{` ⟨*exprow*⟩ `}`	record field updates

- atomic expressions

⟨*atexp*⟩	::=	⟨*scon*⟩	constants
	|	(`op`)? ⟨*longVid*⟩	identifiers
	|	`{`(⟨*exprow*⟩)? `}`	records
	|	`(`⟨exp_1⟩`,`$\cdots$`,`⟨exp_n⟩`)`	tuples ($n \geq 2$)
	|	`()`	unit value
	|	`#`⟨*lab*⟩	record field selector
	|	`[`⟨exp_1⟩`,`$\cdots$`,`⟨exp_n⟩`]`	lists ($n \geq 0$)
	|	`(`⟨exp_1⟩`;`$\cdots$`;`⟨exp_n⟩`)`	sequential execution
	|	`let` ⟨*declList*⟩ `in` ⟨exp_1⟩`;`$\cdots$`;`⟨exp_n⟩ `end`	local declarations
	|	`_sql (`⟨*sql*⟩`)`	SQL query fragments
	|	`(`⟨*exp*⟩`)`	
⟨*exprow*⟩	::=	⟨*lab*⟩ `=` ⟨*exp*⟩ (`,` *exprow*)?	record fields

The definitions for ⟨*cfunty*⟩ is given in Section 19.21 and those for ⟨*sql*⟩ and ⟨*sqlfn*⟩ are given in Chapter 22.

The above hierarchical definition for expressions represents associatibity among expression constructors. The associatibity of infix operator expressions ⟨*infix*⟩ are determined not by syntax but by the infix operator declarations. In the following sections, we first define in the next (19.1) elaboration rules of infix expressions. In the following sections, we define each of expression constructors and their types in the order of associatibity.

19.1 Elaboration of infix expressions

The following infix declamations give identifiers in the ⟨*vid*⟩ class infix operator property.

`infix` (`n`)? ⟨*vidSeq*⟩
`infixr` (`n`)? ⟨*vidSeq*⟩

`infix` defines ⟨*vidSeq*⟩ as left associative infix operators and `infixr` defines ⟨*vidSeq*⟩ as right associative infix operators. Optional integer (n)? (from 0 to 9) specifies association strength (with 9 the strongest). If n is omitted then 0 is assumed. Declaration

`nonfix` ⟨*vidSeq*⟩

cancel infix operator property of identifiers ⟨*vidSeq*⟩.

Infix expressions are converted to applications to tuples according to the association strength.

source	result
⟨exp_1⟩ ⟨*vid*⟩ ⟨exp_2⟩	`op` ⟨*vid*⟩ (⟨exp_1⟩,⟨exp_2⟩)
⟨pat_1⟩ ⟨*vid*⟩ ⟨pat_2⟩	`op` ⟨*vid*⟩ (⟨pat_1⟩,⟨pat_2⟩)

The syntax, when appears in expressions and patters,

`op` ⟨*vid*⟩

cancel the infix status of ⟨*vid*⟩. Therefore if the identifier `foo` has infix status, then the following two code fragments are equivalent.

```
1 foo 2
op foo (1,2)
```

The following are implicitly declared in all the compilation unit and the interactive mode.

```
infix 7 * / div mod
infix 6 + - ^
infixr 5 :: @
infix 4 = <> > >= < <=
infix 3 := o
infix 0 before
```

The above hierarchical syntax with infix declarations determines the association strength of expressions. For example, record update expression (⟨exp_1⟩ `#` `{`⟨*lab*⟩`=`⟨exp_2⟩`}`) associates tightly than strongest infix operators (those that are declared with `infix 9`), and expression constructs `if` ⟨*exp*⟩ `then`⟨*exp*⟩ `else` ⟨*exp*⟩ and others associates weakly than weakest infix operators (those that are declared with `infix 0`.

19.2 Constants ⟨*scon*⟩

Constant expressions (⟨*scon*⟩) are constant literals of types ⟨*int*⟩ , ⟨*word*⟩ , ⟨*real*⟩ , ⟨*string*⟩ , ⟨*char*⟩ defined in Section 17.2. Among them, literals of ⟨*int*⟩ , ⟨*word*⟩ , and ⟨*real*⟩ are overloaded, and their types are determined by their context. If there is no contextual type restriction, then they have the predefined default type.

class	possible types	the default type
⟨*int*⟩	`int, int32, int64, int8, int16, intInf`	`int`
⟨*word*⟩	`word, word32, word64, word8, word16`	`word`
⟨*real*⟩	`real, real32`	`real`

Constant literals ⟨*string*⟩ and ⟨*char*⟩ have `string` type and `char` type, respectively.

These constant literals evaluate to the corresponding values determined in the underlying architecture. The details of atomic value representations are defined in Chapter 29.

The following are simple examples involving overloaded constants.

```
# val one = 1;
val one = 1 : int
# val oneIntInf = 1 : intInf;
val oneIntInf = 1 : intInf
# fun fact 0 = oneIntInf
>   | fact n = n * fact (n - 1);
val fact = fn : intInf -> intInf
# fact 30;
val it = 265252859812191058636308480000000 : intInf
# 0w100;
val it = 0wx64 : word
# "smlsharp";
val it = "smlsharp" : string
# #"S";
val it = #"S" : char
# 3.141592;
val it = 3.141592 : real
```

19.3 Long identifier expression ⟨*longVid*⟩

A long identifier ⟨*longVid*⟩ , appears as an expression, evaluates to the type and dynamic value that the ⟨*longVid*⟩ bound to in the current environment.

⟨*longVid*⟩ is of the form ⟨$strId_1$⟩ . ⋯ . ⟨$strId_n$⟩ . ⟨*vid*⟩ . Structure identifier ⟨$strId_i$⟩ corresponds to a (nested) structure, according to the static scope rule defined in Chapter 21. As defined in Chapter 16, each structure declaration has the effect of extending the current environment with the set of bindings of long identifiers. The set of binding generated by the structure declaration corresponding to ⟨$strId_{i-1}$⟩ contains the binding obtained from the binding corresponding to ⟨$strId_i$⟩ by prefixing the structure name ⟨*strId*⟩ to each of the long identifier.

The type and value of ⟨*longVid*⟩ are therefore the same as those that ⟨*vid*⟩ is bound to in the structure corresponding to ⟨$strId_n$⟩ . If $n = 0$ then the type and value are those that ⟨*vid*⟩ is bound to at the top-level. A simple example is shown below.

```
# structure A = struct val x = 99 end;
structure A =
  struct
    val x = 99 : int
  end
# structure B = struct structure C = A end;
structure B =
  struct
    structure C =
```

```
      struct
        val x = 99 : int
      end
  end
# B.C.x;
val it = 99 : int
```

19.4 Record expression { ⟨lab_1⟩ = ⟨exp_1⟩ , ... , ⟨lab_n⟩ = ⟨exp_n⟩ }

A record expression is a collection of pairs of labels ⟨lab_i⟩ and expressions ⟨exp_i⟩.

A record expression is evaluated as follows. First, it is checked whether all of its labels are distinct. If not, a type error occurs. Second, each expression ⟨exp_i⟩ is evaluated to a type ⟨ty_i⟩ and value v_i in the order of their occurrences in the record expression. Finally, they are sorted in the order of label strings and constitutes a record type

{ ⟨lab'_1⟩ : ⟨ty'_1⟩ , ... , ⟨lab'_n⟩ : ⟨ty'_n⟩ }

and record value

{ ⟨lab'_1⟩ = v'_1 , ... , ⟨lab'_n⟩ = v'_n }

as the type and value of the record expression.

Therefore, for example, when

```
val x = {SML = (print "SML#";"SML#"), IS = (print " is ";" is "), SHARP = (print
" sharp!\n";" sharp!\n")}
```

is evaluated, it prints `"SML# is sharp!"` and then `x` is bound to the type and value like the following

```
val x = {IS = " is ", SHARP = " sharp!\n", SML = "SML#"} : {IS: string, SHARP:
string, SML: string}
```

19.5 Tuple expression (⟨exp_1⟩ , ··· , ⟨exp_n⟩) and unit expression ()

A tuple expression (⟨exp_1⟩ , ··· , ⟨exp_n⟩) is a syntactic sugar that is equivalent to a record expression with numeric labels {1 = ⟨exp_1⟩ , ··· , n = ⟨exp_n⟩}. Before it is evaluated, it is translated to its corresponding record expression. In the interactive session of SML#, the type and value of records whose labels are numerals are printed as tuples.

The `unit` type is the type of empty tuple. Different from the Definition of Standard ML, in SML#, `unit` and the empty record type `{}` are distinct; the empty record type has the empty record kind.

The following shows an example of an interactive session using tuples:

```
# val a = (1, 2);
val a = (1, 2) : int * int
# val b = {1 = 1, 2 = 2};
val b = (1, 2) : int * int
# type foo = {1: int, 2: int}
type foo = int * int
# fun f (x : foo) = (x, x);
# val f = fn : int * int -> (int * int) * (int * int)
val f = fn : int * int -> (int * int) * (int * int)
# f a;
val it = ((1, 2), (1, 2)) : (int * int) * (int * int)
# f b;
val it = ((1, 2), (1, 2)) : (int * int) * (int * int)
# ();
val it = () : unit
# {};
val it = {} : {}
```

19.6 Field selector expression # ⟨*lab*⟩

A field selector is the field selection primitive function of the following type:

```
['a#{⟨lab⟩:'b}, 'b. 'a -> 'b]
```

This type indicates a polymorphic function that takes a record that contains at least ⟨*lab*⟩ field of type `'b` and returns a value of type `'b`. From this type, it is clear that a record field selector can be applied to various record expressions that contains the specific label. The following shows an example:

```
# #y;
val it = fn : ['a#{y: 'b}, 'b. 'a -> 'b]
# #y x = 1, y = 2;
val it = 2 : int
# fun YCord x = #y x;
val YCord = fn : ['a#{y: 'b}, 'b. 'a -> 'b]
# #2;
val it = fn : ['a#{2: 'b}, 'b. 'a -> 'b]
# #2 (1,2,3);
val it = 2 : int
```

A tuple type is a special case of record types and therefore a record field selector is also applicable to tuples.

19.7 List expression [⟨exp_1⟩ , ··· , ⟨exp_n⟩]

A list expression is a sequence of expressions of elements that is separated by commas and surrounded by `[` and `]`. When evaluated, a list expression yields the list type and a value of a list type. The list type is defined as the following data type.

```
infixr 5 ::
datatype 'a list = op :: of 'a * 'a list | nil
```

A list expression is syntactically translated as follows.

source	result
[⟨exp_1⟩ , ··· , ⟨exp_n⟩]	⟨exp_1⟩ :: ··· :: ⟨exp_n⟩ :: nil

$(0 \leq n)$

From this translation, every element expression in a list expression must have the same type τ, and the type of a list expression is *ty* `list`, and the value of the expression is a data of the form `::(`v_1`, ::(`v_2`,··· ::(`v_n`,nil)···))`.

The following shows an example.

```
# [1, 2, 3, 4];
val it = [1, 2, 3, 4] : int list
# [fn x => x, fn x => x + 1];
val it = [fn, fn] : (int -> int) list
# [];
val it = [] : ['a. 'a list]
```

As seen in the example, the empty list, `[]`, has a polymorphic type.

19.8 Sequential execution expression (⟨exp_1⟩ ; ··· ; ⟨exp_n⟩)

This is the expression that executes from ⟨exp_1⟩ to ⟨exp_n⟩ in this order and returns the type and value of the last expression. Evaluation of SML# expressions contains side-effects in general. This expression is usually used to control side-effects. The following shows an example.

```
# val x = ref 1;
val x = ref 1 : int ref
# fun inc () = (x := !x + 1; !x);
val inc = fn : unit -> int
# inc();
val it = 2 : int
# inc();
val it = 3 : int
```

`ref`, `:=`, and `!` in this example are builtin primitives for reference types described in Section 19.20.

19.9 Local declaration expression `let` ⟨*declList*⟩ `in` ⟨exp_1⟩ `;` $\cdots$ `;` ⟨exp_n⟩ `end`

A let expression `let` ⟨*declList*⟩ `in` ⟨exp_1⟩ `;` $\cdots$ `;` ⟨exp_n⟩ `end` allows you to introduce local declarations that are valid only in expressions ⟨exp_1⟩ ; $\cdots$; ⟨exp_n⟩ . This expression is evaluated as follows.

1. Each declaration in ⟨*declList*⟩ is evaluated sequentially and adds the resulting environment to the current environment.
2. Under the augmented environment, expressions ⟨exp_1⟩ ; $\cdots$; ⟨exp_n⟩ are evaluated in this order.
3. The type and value of the last expression is those of this entire expression.

19.10 Function application expression ⟨*appexp*⟩ ⟨*atexp*⟩

Atomic expressions we have introduced so far are the smallest units of expressions that includes delimiters in their own syntax. Expressions are usually constructed by combining those atomic expressions with expression constructors. The most tightly connected expression constructor is function applications. Since SML# inherits the traditions of lambda calculus, the syntax of function applications is ⟨*appexp*⟩ ⟨*atexp*⟩ , which is just a sequence of function expression ⟨*appexp*⟩ and argument expression ⟨*atexp*⟩ . As seen in this syntax, function applications are left-associative. In the last example, `#name f("joe", 21)` is interpreted as `((#name f) ("joe", 21))` and therefore causes a type error.

```
# val f = fn x => fn y => fn z => (x,y,z);;
val f = fn : ['a. 'a -> ['b. 'b -> ['c. 'c -> 'a * 'b * 'c]]]
# f 1 2 (3,4);
val it = (1,2,(3,4)) : int * int * (int * int)
# fun f (x,y) = {name=x,age=y};
val f = fn : ['a, 'b. 'a * 'b -> age: 'b, name: 'a]
# #name (f ("joe", 21));
val it = "joe" : string
# #name f("joe", 21);
(interactive):5.0-5.6 Error:
(type inference 028) operator and operand don't agree
operator domain: 'BCUJ#{name: 'BCUI}
         operand: ['a, 'b. 'a * 'b -> {age: 'b, name: 'a}]
```

19.11 Field update expression ⟨*appexp*⟩ `#` `{` ⟨*exprow*⟩ `}`

This is the expression that produces a new record obtained by updating the record denoted by ⟨*appexp*⟩ with fields specified in `{` ⟨*exprow*⟩ `}`. The type of this expression is same as that of ⟨*appexp*⟩ . The following shows an example.

```
# {x = 1, y = 2} # {x = 2};
val it = {x = 2, y = 2} : {x : int, y : int}
```

As seen in the following example, this expression has polymorphic type with respect to records.

```
# fun incX r = r # {x = #x r + 1};
val incX = fn : ['a#{x: int}. 'a -> 'a]
# incX {x = 1, y = 2};
val it = {x = 2, y = 2} : {x : int, y : int}
```

19.12 Type constraint expression ⟨*exp*⟩ : ⟨*ty*⟩

This expression annotates the type ⟨*ty*⟩ to the expression ⟨*exp*⟩. The type of ⟨*exp*⟩ must be either same as or more general than ⟨*ty*⟩. The following shows an example.

```
# [] : int list;
val it = [] : int list
# 1 : intInf;
val it = 1 : intInf
# fn x => x : int;
val it = fn : int -> int
# fn x => x : 'a -> 'a;
val it = fn : ['a. ('a -> 'a) -> 'a -> 'a]
```

As seen in the last example, ⟨*ty*⟩ may contain type variables. Type variables in a type annotation are never instantiated. Unless there exists an explicit declaration of such type variables, the scope of type variables is the whole of the inner-most `val` declaration in which the type variables occur. See Section 23.1 for the type variable declarations and their scopes.

19.13 Boolean expressions ⟨exp_1⟩ `andalso` ⟨exp_2⟩ and ⟨exp_1⟩ `orelse` ⟨exp_2⟩

The boolean type is predefined as follows:

```
datatype bool = false | true
```

The name of data constructors usually begins with an capital letter, these two constructors are exceptions of such a convention.

For any two expressions ⟨exp_1⟩ and ⟨exp_2⟩ of `bool` type, the following constructs are provided.

- Logical conjunction : ⟨exp_1⟩ `andalso` ⟨exp_2⟩

  It evaluates ⟨exp_1⟩ at first. If the value of ⟨exp_1⟩ is `true`, then it evaluates ⟨exp_2⟩ and returns its value as the result of the entire expression. Otherwise, the value of this expression is `false`. In this case, ⟨exp_2⟩ is not evaluated.

- Logical disjunction : ⟨exp_1⟩ `orelse` ⟨exp_2⟩

  It evaluates ⟨exp_1⟩ at first. If the value of ⟨exp_1⟩ is `false`, then it evaluates ⟨exp_2⟩ and returns its value as the result of the entire expression. Otherwise, the value of this expression is `true`. In this case, ⟨exp_2⟩ is not evaluated.

Since it may ignore the second expressions, these operations are provided as language constructs, not functions.

These syntaxes are left-associative and in the same precedence. Therefore,

```
false andalso true orelse false
```

is interpreted as `(false andalso true) orelse false` and hence is evaluated to `false`.

Other operations on `bool` type are provided as functions. For example,

```
not : bool -> bool
```

19.14 Exception handling expression ⟨*exp*⟩ `handle` ⟨*match*⟩

This expression catches an exception raised during evaluation of expression ⟨*exp*⟩. ⟨*match*⟩ is of the form

$\langle pat_1 \rangle$ `=>` $\langle exp_1 \rangle$
`|` $\cdots$
`|` $\langle pat_n \rangle$ `=>` $\langle exp_n \rangle$

that indicates the sequence of pairs of a exception pattern and expression to be evaluated if the raised exception is matched with the pattern.

Each pattern $\langle pat_i \rangle$ must includes an exception constructor and therefore be of exception type `exn`. The type of every expression $\langle exp_i \rangle$ must be identical to that of $\langle exp \rangle$. This expression is evaluated as follows.

- $\langle exp \rangle$ is evaluated at first. If this evaluation is normally finished, its value is the value of this entire expression.
- If an exception is raised during the evaluation, it tries to match the exception with each pattern from $\langle pat_1 \rangle$ to $\langle pat_n \rangle$. If it a match succeeds, it binds variables in $\langle pat \rangle_i$ to corresponding parameters of the exception object and evaluates $\langle exp_i \rangle$. The value of $\langle exp_i \rangle$ is the value of this entire expression.
- If the exception does not match with any pattern, the exception is propagated to the outer expression surrounding this expression.

19.15 Exception expression `raise` $\langle exp \rangle$

This expression raises the given exception. $\langle exp \rangle$ must be of exception type `exn`. The type of this expression depends on the context; if there is no restriction on types, this exception has a polymorphic type `'a`.

This expression evaluates $\langle exp \rangle$ to an exception and raise it. Therefore, this expression does not have any value.

19.16 Conditional expression `if` $\langle exp_1 \rangle$ `then` $\langle exp_2 \rangle$ `else` $\langle exp_3 \rangle$

This is for conditional branches. $\langle exp_1 \rangle$ must be of `bool` type. The type of $\langle exp_2 \rangle$ and $\langle exp_3 \rangle$ must be same. Under these restrictions, this entire expression have the same type as $\langle exp_2 \rangle$.

This is evaluated as follows. $\langle exp_1 \rangle$ is evaluated at first. If its value is `true`, $\langle exp_2 \rangle$ is evaluated and the resulting value is the value of the entire expression. Otherwise, $\langle exp_3 \rangle$ is evaluated and the resulting value is the value of the entire expression.

19.17 While expression `while` $\langle exp_1 \rangle$ `do` $\langle exp_2 \rangle$

This is for imperative iterations. $\langle exp_1 \rangle$ must be of `bool` type. The type of this expression is `unit`. This expression iterates the evaluation of $\langle exp_2 \rangle$ while $\langle exp_1 \rangle$ is evaluated to `true`. If $\langle exp_1 \rangle$ is evaluated to `false`, it returns `()`.

19.18 Case expression `case` $\langle exp \rangle$ `of` $\langle match \rangle$

This expresses general conditional branches on the value of $\langle exp \rangle$. $\langle match \rangle$ is of the form

$\langle pat_1 \rangle$ `=>` $\langle exp_1 \rangle$
`|` $\cdots$
`|` $\langle pat_n \rangle$ `=>` $\langle exp_n \rangle$

where each $\langle pat_i \rangle$ is a pattern containing data constructors and $\langle exp_i \rangle$ is the expression to be evaluated if $\langle pat_i \rangle$ is matched against the value of $\langle exp \rangle$.

Each pattern $\langle pat_i \rangle$ is a data pattern consisting of constructors and variables, as described in Chapter 20. This set of patterns must satisfy the following restrictions:

- All variables occurring in a pattern must be distinct.
- Every pattern must have the same type as the type of $\langle exp \rangle$.

- For any i such that 2 $lei \leq n$, ⟨pat_i⟩ must not be redundant. In other words, the set of data covered by ⟨pat_i⟩ must not be a subset of the union of the sets of data covered by ⟨pat_1⟩, ..., ⟨pat_{i-1}⟩.

For example, the following violates the third rule.

```
# fn x => case x of (X, 1, 2) => 1 | (1, X, 2) => 2 | (1, 1, 2) => 3;
(interactive):6.8-6.65 Error: match redundant and nonexhaustive
      (X, 1, 2) => ...
      (1, X, 2) => ...
  --> (1, 1, 2) => ...
```

Under these restrictions, this entire expression has the same type as each expression ⟨exp_i⟩.

This case expression is evaluated as follows. It tries to match the value of ⟨*exp*⟩ with each pattern from ⟨pat_1⟩ to ⟨pat_n⟩ in this order. For the first matched pattern ⟨pat_i⟩, it binds variables in ⟨pat_i⟩ to their corresponding part of the value and evaluates ⟨exp_i⟩ to the result value. If no pattern matches with the value, it raises `Match` exception.

19.19 Function expression `fn` ⟨*match*⟩

This is for the anonymous function indicated by ⟨*match*⟩. ⟨*match*⟩ is of the form

```
  ⟨pat1⟩ => ⟨exp1⟩
| ...
| ⟨patn⟩ => ⟨expn⟩
```

that indicates pairs of an argument pattern and its corresponding function body.

The type of each pair of a pattern and expression ⟨pat_i⟩ =¿ ⟨exp_i⟩ is calculated as follows: first, the type ⟨ty_i⟩ of ⟨pat_i⟩ is calculated; second, under the environment extended with variables in the pattern, the type ⟨ty'_i⟩ of ⟨exp_i⟩ is calculated; and finally, the two types are combined into a function type ⟨ty_i⟩ `->` ⟨ty'_i⟩. The type of this entire expression is ⟨*ty*⟩ `->` ⟨*ty'*⟩ that is obtained by unifying the types of the matching pairs. The value of this function is the function closure consisting of the current value environment and the function expression itself.

19.20 Builtin types and builtin primitives

Among the set of built-in types, exception type (`exn`) is manipulated by `handle` and `raise` expressions defined in section 19.14. There is no primitive functions for the unit type (`unit`). All the other built-in types are manipulated by built-in primitive functions through function application.

Values of built-in types other than reference type (`'a ref`) and array types (`'a array`) are ordinary values in a standard functional language semantics, and their built-in primitive functions are those that take values and return values.

Values of reference types (`'a ref`) and array types (`'a array`) are pointers to mutable memory blocks, and their built-in primitive function may have side effect of destructive memory update.

For values of type ⟨*ty*⟩ `ref`, the following built-in primitive functions are provided.

```
ref : ⟨ty⟩ -> ⟨ty⟩ ref
  ! : ⟨ty⟩ ref -> ⟨ty⟩
 := : ⟨ty⟩ ref * ⟨ty⟩ -> unit
infix 3 :=
```

`ref` ⟨*exp*⟩ creates a reference, i.e. a pointer to a memory block containing a value denoted by ⟨*exp*⟩. `!` ⟨*exp*⟩ dereferences the pointer and return the stored value. ⟨exp⟩$_1$ `:=` ⟨exp⟩$_2$ destructively updates the memory block pointed to by the denotation of ⟨exp⟩$_1$ with the value denoted by ⟨exp⟩$_2$.

The built-in primitive functions for any other types, including ⟨*ty*⟩ `array`, are provided by the following structures of the library defined in Chapter 25. `General` structure includes the reference primitives and defines exceptions that may raised by built-in primitives.

builtin type	structure	signature
`int8`	Int8	INTEGER
`int16`	Int16	INTEGER
`int`	Int, Int32	INTEGER
`int64`	Int64	INTEGER
`intInf`	IntInf	INTINF
`word8`	Word8	WORD
`word16`	Word16	WORD
`word`	Word, Word32	WORD
`word64`	Word64	WORD
`real32`	Real32	REAL
`real`	Real, Real64	REAL
`char`	Char	CHAR
`string`	String	STRING
τ `array`	Array	ARRAY
τ `vector`	Vector	VECTOR
τ `ref`, `exn`	General	GENERAL

The set of built-in primitive functions for a built-in type can be displayed by evaluating a structure replication declaration of the structure that implements the built-in type, as seen in the following example.

```
# structure X = Int;
structure X =
  struct
    type int = Int32.int
    val * = <builtin> : int * int -> int
    val + = <builtin> : int * int -> int
    val - = <builtin> : int * int -> int
    val < = <builtin> : int * int -> bool
    val <= = <builtin> : int * int -> bool
    val > = <builtin> : int * int -> bool
    val >= = <builtin> : int * int -> bool
    val abs = <builtin> : int -> int
    val compare = fn : int * int -> General.order
    val div = <builtin> : int * int -> int
    val fmt = fn : StringCvt.radix -> int -> string
    val fromInt = fn : int -> int
    val fromLarge = fn : intInf -> int
    val fromString = fn : string -> int option
    val max = fn : int * int -> int
    val maxInt = SOME 2147483647 : int option
    val min = fn : int * int -> int
    val minInt = SOME  2147483648 : int option
    val mod = <builtin> : int * int -> int
    val precision = SOME 32 : int option
    val quot = <builtin> : int * int -> int
    val rem = <builtin> : int * int -> int
    val sameSign = fn : int * int -> bool
    val scan = fn
      : ['a. StringCvt.radix
         -> ('a -> (char * 'a) option) -> 'a -> (int * 'a) option]
    val sign = fn : int -> int
    val toInt = fn : int -> int
    val toLarge = fn : int -> intInf
    val toString = fn : int -> string
    val ~ = <builtin> : int -> int
  end
```

19.21 Static import expression: _import ⟨string⟩ : ⟨cfunty⟩

This expression allows you to use C functions as SML# functions. The meaning of each component is as follows:

- ⟨*string*⟩ : the name of the C function. This is the name that linker recognizes as an external symbol name.
- ⟨*cfunty*⟩ : the type of the C function. It must be of the following form.

C function type: ⟨*cfunty*⟩

⟨*cfunty*⟩	::=	(⟨*cfunattr*⟩)? ⟨*argTyList*⟩ `->` ⟨*retTyOpt*⟩	
⟨*argTyList*⟩	::=	`(` ⟨*argTy*⟩ `,...,` ⟨*argTy*⟩ (`,` ⟨*varArgs*⟩)? `)`	(multiple arguments)
	|	⟨*argTy*⟩	(only one argument)
	|	`()`	(no argument)
⟨*retTyOpt*⟩	::=	⟨*retTy*⟩	
	|	`()`	(`void` type in C)

callback function type: ⟨*argfunty*⟩

⟨*argfunty*⟩	::=	(⟨*cfunattr*⟩)? ⟨*retTylist*⟩ `->` ⟨*argTyOpt*⟩	
⟨*retTyList*⟩	::=	`(` ⟨*retTy*⟩ `,...,` ⟨*retTy*⟩ (`,` ⟨*varRets*⟩)? `)`	(multiple arguments)
	|	⟨*retTy*⟩	(only one argument)
	|	`()`	(no argument)
⟨*argTyOpt*⟩	::=	⟨*argTy*⟩	
	|	`()`	(`void` type in C)

interoperable type: ⟨*interoperableTy*⟩

⟨*interoperableTy*⟩	::=	(⟨*tySeq*⟩)? ⟨*longTycon*⟩	(interoperable types only. See below for details)

argument type from SML# to C: ⟨*argTy*⟩

⟨*argTy*⟩	::=	⟨*argTy*⟩ `*` ⋯ `*` ⟨*argTy*⟩	(types of pointers to a structure)
	|	`{` ⟨*argTyRow*⟩ `}`	(types of pointers to a structure)
	|	⟨*tyvar*⟩	(`boxed` kind only)
	|	⟨*interoperableTy*⟩	
	|	⟨*argfunty*⟩	(callback function argument types)
⟨*argTyRow*⟩	::=	⟨*lab*⟩ `:` ⟨*argTy*⟩ (`,` ⟨*argTyRow*⟩)?	(⟨*lab*⟩ must begin with ⟨*decimal*⟩)

return type from C to SML#: ⟨*retTy*⟩

⟨*retTy*⟩	::=	⟨*interoperableTy*⟩	
	|	⟨*tyvar*⟩	(`boxed` kind only)

variable-length argument type specification: ⟨*varArgs*⟩ and ⟨*varRets*⟩

⟨*varArgs*⟩	::=	`...` `(` ⟨*argTy*⟩ `,...,` ⟨*argTy*⟩ `)`
⟨*varRets*⟩	::=	`...` `(` ⟨*retTy*⟩ `,...,` ⟨*retTy*⟩ `)`

C function attributes: ⟨*cfunattr*⟩

⟨*cfunattr*⟩	::=	`__attribute__((` ⟨*attr*⟩ `,...,` ⟨*attr*⟩ `))`
⟨*att*⟩	::=	`cdecl` | `stdcall` | `fastcc` | `pure` | `fast`

⟨*cfunty*⟩ indicates the type of the C function in SML#'s type names and type notation. ⟨*interoperableTy*⟩, the type names for C functions, must satisfy both of the following:

1. ⟨*interoperableTy*⟩ は以下のいずれかでなければならない.
   - Interoperable atomic types: `int`, `int8`, `int16`, `int64`, `word`, `word8`, `word16`, `word64`, `real`, `real32`, `char`, or `string`.
   - The types of C pointers: `codeptr`, ⟨*interoperableTy*⟩ `ptr`, or `unit ptr`.
   - The types of C pointers: ⟨*tyvar*⟩ `ptr` (ただし ⟨*tyvar*⟩ must be of either `boxed` or `unboxed` kind).

- The types of size: ⟨*ty*⟩ `size`.
- Array types: ⟨*interoperableTy*⟩ `array`, ⟨*interoperableTy*⟩ `vector`, or ⟨*interoperableTy*⟩ `ref`.
- Polymorphic array types: ⟨*tyvar*⟩ `array`, ⟨*tyvar*⟩ `vector`, or ⟨*tyvar*⟩ `ref` (ただし ⟨*tyvar*⟩ must be of either `boxed` or `unboxed` kind).
- An alias of one of the above types defined by a `type` declaration. (⟨*tySeq*⟩)? ⟨*longTycon*⟩ must be expanded to one of the above types. If an alias type occurs as an ⟨*interoperableTy*⟩ in the context of ⟨*argTy*⟩ , (⟨*tySeq*⟩)? ⟨*longTycon*⟩ may be expanded to an tuple or record type that can be regarded as an ⟨*argTy*⟩ .

2. Neither `string`, `array`, `vector`, nor `ref` may occur as the type of values that would be passed from C to SML# or be overwritten by a C function. In other words, these types must not occur as
   - ⟨*interoperableTy*⟩ in the context of ⟨*retTy*⟩ , and
   - the type parameter of `array`, `ref`, and `ptr` type.

以上の条件を満たす ⟨*interoperableTy*⟩ は，その型名から自然に類推される C の型に相当する．対応を以下に示す．

⟨*interoperableTy*⟩	対応する C の型
`int` and its family	signed integer of the same size
`word` and its family	unsigned integer of the same size
`real`	`double`
`real32`	`float`
`char`	`char`
`string`	`const char *`
`codeptr`	pointer to a function
τ `ptr`	pointer to τ
`unit ptr`	`void *` or pointer to an incomplete type
τ `size`	`size_t`
τ `array`	pointer to the beginning of an array of τ
τ `vector`	pointer to the beginning of a vector of τ
τ `ref`	pointer to an one-element array of τ

Note that the size of `int` and `word` is always 32 bits. In almost of all modern operating systems, SML#'s `int` is identical to C's `int`, whereas they are not identical in some systems whose `int` is not 32 bits.

⟨$argTy_1$⟩ `*` $\cdots$ `*` ⟨$argTy_n$⟩ and `{` ⟨lab_1⟩ `:` ⟨$argTy_1$⟩ `,` $\cdots$, ⟨lab_n⟩ `:` ⟨$argTy_n$⟩ `}` of ⟨*argTy*⟩ corresponds to the type of pointers to a `const` structure whose members are of type ⟨$argTy_1$⟩, $\cdots$, ⟨$argTy_n$⟩ in the order of ⟨*decimal*⟩ of labels. If all $\langle argTy \rangle_i$ are identical, it also corresponds to a pointer to the beginning of an `const` array of n elements of $\langle argTy \rangle_i$.

If and only if a C function acts like a parametric polymorphic function from the perspective of SML# programs, it is allowed to use type variables as a C function's argument or return type. For example, an identical function in C

```
void *id(void *x) { return x; }
```

is allowed to be imported like this:

```
val 'a#boxed id = _import "id" : 'a -> 'a
```

Another example is `printf` function that prints an arbitrary pointer value.

```
val 'a#boxed printPtr = _import "printf" : (string,...('a)) -> int
```

The following C function attributes are available:

`cdecl` The C function follows the standard calling convention of C functions on the target platform. This is the default if no calling convention is specified through function attributes.

`stdcall` The C function follows stdcall calling convention on Windows platforms.

`fastcc` The C function follows fastcc calling convention provided by LLVM.

`pure` The C function is *pure* in the sense of SML#. In other words, C functions of this attribute does not perform any memory update and I/O, and its return value is decided only from the list of arguments. This attribute affects the optimization of the SML# compiler.

`fast` The C function returns very quickly. It neither execute SML# code through callbacks nor pause the thread execution. The SML# compiler generates efficient invocation code for such functions. Note that, if a C function of this attribute either consumes much time or pause a thread execution, garbage collection would be suspended and consequently all threads would be suspended.

The type of this expression is the SML# function type corresponding to the type specified in ⟨*cfunty*⟩. The correspondence is defined as follows.

C function type	SML# function type
`(` ⟨*argTy*⟩$_1$, ..., ⟨*argTy*⟩$_n$ (⟨*varArgs*⟩)? `)` `->` ⟨*retTy*⟩	⟨*argTy*⟩$_1$ * ⋯ * ⟨*argTy*⟩$_n$ (* ⟨*varArgs*⟩)? `->` ⟨*retTy*⟩

`()` in the argument or return type appear as `unit` in the SML# type. ⟨*interoperableTy*⟩, ⟨*tyvar*⟩, and ⟨***⟩ appear in the SML# type without modification. ⟨*argfunty*⟩ is interpreted similarly.

The value of this expression is the SML# function that calls the C function specified in ⟨*string*⟩. As long as the C function type specification is correct, this function can be used similarly to ordinary SML# functions.

19.22 Dynamic import expression: ⟨*exp*⟩ : _import ⟨*cfunty*⟩

Similarly to the static import expression, this imports a function pointer obtained by dynamic linking as an SML# function. The type of ⟨*exp*⟩ must be `codeptr`. When evaluating this expression, ⟨*exp*⟩ is evaluated to an function pointer. If the function pointer points to a C function of the type indicated by ⟨*cfunty*⟩, the value of this expression is the SML# function that calls the C function. Otherwise, the value of this expression is undefined.

19.23 Size expression _sizeof(⟨*ty*⟩)

This is a constant expression denoting the size (in bytes) of the values of type ⟨*ty*⟩. The type of this expression is ⟨*ty*⟩ `size` and its value is an integer indicating the number of bytes.

The size expressions are usually used to call polymorphic C functions. For example, the following is the import expression that imports `memcpy` in the C standard library for copying the first element of arrays:

```
val 'a#unboxed memcpy =
    _import "memcpy" : ('a array, 'a vector, 'a size) -> unit ptr
```

This function is called in the following way:

```
fun 'a#unboxed copy (a : 'a array, v) =
    if Array.length a > 0 andalso Vector.length v > 0
    then memcpy (a, v, _sizeof('a))
    else ()
```

19.24 Dynamic type cast expression _dynamic ⟨*exp*⟩ as ⟨*ty*⟩

This expression performs dynamic type cast of dynamically-typed value ⟨*exp*⟩ to ⟨*ty*⟩. The type of ⟨*exp*⟩ must be τ `Dynamic.dyn` for some τ. The type of this expression is ⟨*ty*⟩.

To evaluate this expression, ⟨*exp*⟩ is evaluated and the dynamically-typed value v is obtained. The value of this expression depends on the structure of v and ⟨*ty*⟩. The rule of dynamic type cast is the following:

- When ⟨*ty*⟩ is `Dynamic.void Dynamic.dyn`, the value of this expression is v.
- When ⟨*ty*⟩ is τ `Dynamic.dyn` for some *tau*, if v has a view of τ (a substructure of v can be extracted as a value of τ), the value of this expression is v. Otherwise, the `Dynamic.RuntimeTypeError` excepstion is raised.

- When ⟨*ty*⟩ does not include `Dynamic.dyn`, the type of v is identical to ⟨*ty*⟩, the value of this expression is the value obtained by type-casting v to ⟨*ty*⟩. Otherwise, the `Dynamic.RuntimeTypeError` excepstion is raised.
- When ⟨*ty*⟩ includes `Dynamic.dyn` as its substructure, the above rules are applied recursively on the structure of v and ⟨*ty*⟩.

For example, suppose the following list of records:

```
val r = Dynamic.dynamic [{name = "Joe", age = 21}, {name = "Sue", age = 31}];
```

The following casts are correct:

```
_dynamic r as {name:string, age:int} list;
_dynamic r as {name:Dynamic.void Dynamic.dyn, age:int} list;
_dynamic r as Dynamic.void Dynamic.dyn;
_dynamic r as Dynamic.void Dynamic.dyn list;
_dynamic r as {name:string, age:int} Dynamic.dyn list;
_dynamic r as {name:string} Dynamic.dyn list;
_dynamic r as {name:string, age:int} list Dynamic.dyn;
_dynamic r as {age:int} list Dynamic.dyn;
```

Note that v is not always a data structure that is typable in ML. For example, suppose the following heterogeneous list:

```
val l = Dynamic.fromJson
          "[{\"name\":\"Joe\", \"age\":21},\
          \{\"name\":\"Sue\", \"grade\":2.0},\
          \{\"name\":\"Robert\", \"nickname\":\"Bob\"}]";
```

The following casts are correct:

```
_dynamic l as Dynamic.void Dynamic.dyn;
_dynamic l as Dynamic.void Dynamic.dyn list;
_dynamic l as {name:string} Dynamic.dyn list;
_dynamic l as {name:string} list Dynamic.dyn;
_dynamic l as {name:Dynamic.void Dynamic.dyn} Dynamic.dyn list;
```

To use this expression in the separate compilation mode, `"reify.smi"` must be `_require`d.

19.25 Case branch expression with dynamic type cast `_dynamiccase` ⟨*exp*⟩ `of`

This expresses general conditional branches on the dynamically-typed value of ⟨*exp*⟩. ⟨*match*⟩ is of the form:

```
  ⟨pat_1⟩ => ⟨exp_1⟩
| ...
| ⟨pat_n⟩ => ⟨exp_n⟩
```

The type of each pattern may differ from each other. Variable patterns and anonymous patterns occurring in ⟨pat_i⟩ must be type-annotated. In addition, the set of patterns must satisfy the following restriction:

- When categorizing the set of patterns by their types, for each set of patterns of the same type, the set of patterns must satisfy the same condition as the `case` expression.

For example, the following is redundant on `int` patterns and therefore violates the rule.

```
# fn x => _dynamiccase x of x:int => "int" | x:real => "real" | 0:int => "zero";
(interactive):1.8-1.76 Error: match redundant
      x => ...
  --> 0 => ...
```

This expression is evalated as follows. It tries to match the dynamically-typed value v of $\langle exp \rangle$ with each pattern from $\langle pat_1 \rangle$ to $\langle pat_n \rangle$ in this order. Before trying matching with a pattern, v is casted to the type of the pattern in the same way as the `_dynamic` expression. For the first matched pattern $\langle pat_i \rangle$, it binds variables in $\langle pat_i \rangle$ to their corresponding part of the value and evaluates $\langle exp_i \rangle$ to the result value. If no dynamic type cast succeeds, it raises the `Dynamic.RuntimeTypeError` exception. If no pattern matches with the value, it raises the `Match` exception.

To use this expression in the separate compilation mode, `"reify.smi"` must be `_require`d.

Chapter 20

Patterns and Pattern Matching

Patterns ⟨*pat*⟩ describe structures of values, and are used to bind variables values through pattern matching mechanism, built-in val-bind declarations (`valBind`)23.1, function declarations (`funDecl`)23.2, `case` expressions, `fn` expression, and `handle` expressions.

When evaluated, pattern matching generates static type environment at compile time, and dynamic value environment at runtime. These environments are used to added to the evaluation environment for the expressions and declarations in the scope of the pattern.

Syntax of Patterns is given below.

- top-level pattern

⟨*pat*⟩	::=	⟨*atpat*⟩	
	|	(`op`)? ⟨*longVid*⟩ ⟨*atpat*⟩	data structure
	|	⟨*pat*⟩ ⟨*vid*⟩ ⟨*pat*⟩	data structure (infix operator form)
	|	⟨*pat*⟩ `:` ⟨*ty*⟩	pattern with type constraint
	|	⟨*vid*⟩ (`:` ⟨*ty*⟩)? `as` ⟨*pat*⟩	layered pattern

- atomic patterns

⟨*atpat*⟩	::=	⟨*scon*⟩	constant
	|	`_`	anonymous pattern
	|	⟨*vid*⟩	variable and constructor
	|	⟨*longVid*⟩	constructor
	|	`{`(⟨*patrow*⟩)? `}`	record pattern
	|	`()`	`unit` type constant
	|	`(` ⟨pat_1⟩ `,` $\cdots$ `,` ⟨pat_n⟩ `)`	tuples ($n \geq 2$)
	|	`[` ⟨pat_1⟩ `,` $\cdots$ `,` ⟨pat_n⟩ `]`	list ($n \geq 0$)
	|	`(` ⟨*pat*⟩ `)`	
⟨*patrow*⟩	::=	`...`	anonymous field
	|	⟨*lab*⟩ `=` ⟨*pat*⟩ (`,` ⟨*patrow*⟩)?	record fields
	|	vid(`:` ⟨*ty*⟩)? (`as` ⟨*pat*⟩)? (`,` ⟨*patrow*⟩)?	label and variable

The following subsections define for each pattern ⟨*pat*⟩, its type ⟨*ty*⟩, matching values, and the resulting type environment. The dynamic environment generated at runtime corresponds to the generated type environment and it binds identifiers to the matched values.

Constant pattern: ⟨*scon*⟩ They are the same the constant expressions defined in 19.2. They have the corresponding types, match the same constant values, and generate the empty type environment.

Anonymous pattern:`_` This has arbitrary type determined by the context, matches any value, and generates the empty type environment.

identifier pattern: ⟨*vid*⟩ If the identifier is defined as a constructor, then it has the type of the constructor, matches the constructor, and generate the empty type environment.

If the identifier is not defined or defined as a variable, then it has arbitrary type ⟨*ty*⟩ determined by the context, matches any value of type ⟨*ty*⟩, and generate the type environment { ⟨*vid*⟩ : ⟨*ty*⟩ }.

The following shows simple examples.

```
SML# 4.2.0 (2025-03-24) for x86_64-pc-linux-gnu with LLVM 20.1
# val A = 1
val A = 1 : int
# fn A => 1;
val it = fn : ['a. 'a -> int]
# datatype foo = A;
datatype foo = A
# fn A => 1;
val it = fn : foo -> int
# datatype foo = A of int;
datatype foo = x of int
# fn A => 1;
(interactive):12.3-12.3 Error:
   (type inference 039) data constructor A used without argument in pattern
```

long identifier: ⟨*longVid*⟩ If the long identifier is defined as a constructor, then it has the type of the constructor, matches the constructor, and generate the empty type environment. It is an error if the identifier is not defined as a constructor or defined as a constructor with an argument. The following shows simple examples.

```
# structure A = struct datatype foo = A | B of int val C = 1 end;
structure A =
  struct
    datatype foo = A | B of int
    val C = 1 : int
  end
# val f = fn A.A => 1;
val f = fn : A.foo -> int
# val g = fn A.B => 1;
(interactive):3.11-3.13 Error:
  (type inference 046) data constructor A.B used without argument in pattern
# val h = fn A.C => 1;
(interactive):4.11-4.13 Error: (name evaluation "020") unbound constructor: A.C
```

`A.B` is a constructor with an argument and `A.C` is a variable, function declarations `g` and `h` result in errors.

record pattern: ⟨`{`(⟨*patrow*⟩)? `}`⟩ The pattern `{}` without record fields has the `unit` type, matches the value `()`, and generate the empty type environment.

If it has of the form ⟨*lab*$_1$⟩ : ⟨*ty*$_1$⟩ `,` ..., ⟨*lab*$_n$⟩ : ⟨*ty*$_n$⟩ , with non-empty record fields, there are two cases according to the record field pattern ⟨*patrow*⟩. Let Γ be the type environment generated by the set of patterns in the fields.

1. Monomorphic record pattern, i.e. the case where the anonymous field `...` is not contained. It has the (monomorphic) record type `{` ⟨*lab*$_1$⟩ : ⟨*ty*$_1$⟩ `,` ..., ⟨*lab*$_n$⟩ : ⟨*ty*$_n$⟩ `}`, matches any records containing all the fields that matches the record field patterns ⟨*patrow*⟩, and generate Γ.

2. Polymorphic record pattern, i.e. the case where the anonymous field `...` is contained. It has the polymorphic record type `'a#{` ⟨*lab*$_1$⟩ : ⟨*ty*$_1$⟩ `,` ..., ⟨*lab*$_n$⟩ : ⟨*ty*$_n$⟩ `}` with the record kind of the field types of ⟨*patrow*⟩, matches any records containing all the fields that matches the record field patterns ⟨*patrow*⟩, and generate Γ. The polymorphic type of this entire pattern can be instantiated with any record types having the kind.

record field pattern: ⟨*patrow*⟩

- Anonymous field pattern : `...` It indicates that matching records may contains more fields than the specified fields.
- Field pattern : ⟨*lab*⟩ `=` ⟨*pat*⟩ (`,` ⟨*patrow*⟩)?. Let the filed type and the generated type environment of (`,` ⟨*patrow*⟩)? be ⟨*lab*$_1$⟩ : ⟨*ty*$_1$⟩ ,..., ⟨*lab*$_n$⟩ : ⟨*ty*$_n$⟩ , and Γ. Let the type and the generated type environment of the pattern ⟨*pat*⟩ be ⟨*ty*⟩ and Γ$_0$. If the label ⟨*lab*⟩ is different than any labels ⟨*lab*$_1$⟩ ,..., ⟨*lab*$_n$⟩ , then the type and the generated type environment of the entire pattern are ⟨*lab*⟩ : ⟨*ty*⟩ , ⟨*lab*$_1$⟩ : ⟨*ty*$_1$⟩ ,..., ⟨*lab*$_n$⟩ : ⟨*ty*$_n$⟩ , and Γ$_0$ ∪ Γ. It is a type error if the label ⟨*lab*⟩ is one of ⟨*lab*$_1$⟩ ,..., ⟨*lab*$_n$⟩ .
- variable as label pattern : vid(`:` ⟨*ty*⟩)? (`as` ⟨*pat*⟩)? (`,` ⟨*patrow*⟩)?.

  This is converted to the following field pattern before evaluation.

  vid = vid(`:` ⟨*ty*⟩)? (`as` ⟨*pat*⟩)? (`,` ⟨*patrow*⟩)?.

  The generated static environment and the matching dynamic value are the same as the transformed record pattern. The following shows simple examples.

```
# val f = fn {x:int as 1, y} => x + y;
(interactive):33.8-33.34 Warning: match nonexhaustive
  {x = x as 1, y = y} => ...
val f = fn : {x: int, y: int} -> int
# val g = fn {x = x:int as 1, y = y} => x + y;
(interactive):34.8-34.37 Warning: match nonexhaustive
  {x = x as 1, y = y} => ...
val g = fn : {x: int, y: int} -> int
# f {x = 1, y = 2};
val it = 3 : int
# g {x = 1, y = 2};
val it = 3 : int
```

Tuple pattern: `(` ⟨*pat*$_1$⟩ ,⋯, ⟨*pat*$_n$⟩ `)` This is converted to the monomorphic record pattern

{1= ⟨*pat*$_1$⟩ ,⋯,n= ⟨*pat*$_n$⟩ }.

The generated static environment and the matching dynamic value are the same as the transformed record pattern. The following shows simple examples.

```
# val f = fn (x,y) => 2 * x + y;
val f = fn : int * int -> int
# val g = fn {1=x, 2=y} => 2 * x + y;
val g = fn : int * int -> int
# f 1=1, 2=2;
val it = 4 : int
# g (1,2);
val it = 4 : int
```

List pattern: `[` ⟨*pat*$_1$⟩ ,⋯, ⟨*pat*$_n$⟩ `]` This is converted to the following nested list data structure pattern:

⟨*pat*$_1$⟩ :: ⋯ :: ⟨*pat*$_n$⟩ :: nil

The generated static environment and the matching dynamic value are the same as the transformed constructor application pattern. The following shows simple examples.

```
# val f = fn [x,y] => 2 * x + y;
(interactive):24.8-24.26 Warning: match nonexhaustive
  :: (x, :: (y, nil )) => ...
val f = fn : int list -> int
```

```
# val g = fn (x::y::nil) => 2 * x + y;
(interactive):25.8-25.32 Warning: match nonexhaustive
  :: (x, :: (y, nil )) => ...
val g = fn : int list -> int
# f (1::2::nil);
val it = 4 : int
# g [1,2];
val it = 4 : int
```

Data structure pattern:(`op`)? ⟨*longVid*⟩ ⟨*atpat*⟩ If ⟨*longVid*⟩ is bound to a constructor C of type of the form ⟨ty_1⟩ `->` ⟨ty_2⟩, then it has type ⟨ty_2⟩ and generates a type constructor generated by ⟨*atpat*⟩. This pattern matches a data structure of the form $C(v)$ if the subterm v matches ⟨*atpat*⟩.

Data structure patter in infix form ⟨pat_1⟩ ⟨*vid*⟩ ⟨pat_2⟩ is syntactically converted to `op` ⟨*vid*⟩ `(` ⟨pat_1⟩ `,` ⟨pat_2⟩ `)` before evaluation. The type and type environment being generated and the matching dynamic value are the same as those of converted pattern.

Typed pattern: ⟨*pat*⟩ : ⟨*ty*⟩ If pattern ⟨*pat*⟩ has type ⟨ty_1⟩ ad type environment Γ_0, and ⟨ty_1⟩ and ⟨*ty*⟩ unifies under type substitution S, then this pattern has type $S($⟨*ty*⟩$)$ and type environment $S(\Gamma_0)$. This pattern matches a value of type $S($⟨*ty*⟩$)$ that matches pattern ⟨*pat*⟩.

Layered pattern: ⟨*vid*⟩ (`:` ⟨*ty*⟩)? `as` ⟨*pat*⟩ If the pattern ⟨*pat*⟩ : ⟨*ty*⟩ has type ⟨*ty′*⟩ and a type environment Γ then this pattern has type ⟨*ty′*⟩ and a type environment $\Gamma \cup \{x : \langle ty' \rangle\}$. It matches a dynamic value that the pattern ⟨*pat*⟩ : ⟨*ty*⟩ matches.

Chapter 21

Scope rules for identifier

This chapter defines the static scope rules for names used in programs.

As defined in Section 17.2, identifiers are used as names of the following seven classes.

identifier	name class
⟨*vid*⟩	variable and constructor names
⟨*strid*⟩	structure name
⟨*sigid*⟩	signature name
⟨*funid*⟩	functor name
⟨*tycon*⟩	type constructor name
⟨*tyvar*⟩	type variable name
⟨*lab*⟩	record label

Among them, type variable names have the syntax `'...`, and are distinguished from the other classes of names. All the other names overlap one another. For example, `A` can be used as a name of any of the classes other than type variable name. As defined in Chapter 21, the syntax is defined so that the name class of identifier occurrence is uniquely determined. Furthermore, these name classes are managed as separate name spaces, and same identifier can be used as names of different classes. The following example shows usage of the same identifier `A` as different names.

```
(* 1 *) val A = {A = 1}
(* 2 *) type 'A A = {A: 'A}
(* 3 *) signature A = sig val A : int A end
(* 4 *) functor A () : A = struct val A = {A = 1} end
(* 5 *) structure A : A = A()
(* 6 *) val x = A.A
(* 7 *) val y = A : int A
```

The first occurrence of `A` in line 7 refers to the variable defined in line 1 and the second occurrence of `A` refers to the type constructor defined in line 2. The first occurrence of `A` in line 6 refers to the structure defined in line 5.

Names other than record labels are defined by program constructs and referenced. The following are the program constructs involving name definitions.

1. **Interface files in separate compilation**. ProvideList ⟨*provideList*⟩ in the interface file define names. For example, the interface file containing the following declarations

   ```
   _require "myLibrary.smi"
   val x : int
   datatype foo = A of int | B of bool
   ```

   defines variable name `x`, data constructor names `A,B`, and type constructor name `foo`.

2. **Structure declaration**. The structure name and the set of long names obtained from the set of long names defined in the structure by prefixing the structure name are defined. For example, if the structure S define a set of long names L then a declaration `structure S =` S defines the structure name `S` and the set of long names $\{S.path \mid path \in L\}$.

3. **Functor declaration**. The functor name and the names in the argument specification are defined. For example, `functor F(type foo) =` $\cdots$ defines functor name `F` and type constructor name `foo`.

4. **Type alias declaration**. The type constructor name and the argument type variable names are defined. For example, `type foo =` $\cdots$ defines type constructor (without type parameter) name `foo`.

5. **Datatype declaration**. The type constructor name, the argument type variable names and the data constructor names are defined. For example, `datatype 'a foo = A of int | B of bool` defines type constructor name `foo`, argument type variable name `'a`, and data constructor names `A, B.` .

6. **Exception declarations**. Exception constructor names are defined. For example, `exception E` defines exception constructor (without argument) name `E`.

7. **Val declaration.** The variables occurring in the bound pattern are defined. For example, if `x` is not defined as a data constructor, `val x = 1` defines variable name `x`.

8. **Function declaration.** Function names and variables occurring in argument patterns are defined. For example, if `x` is not defined as a data constructor in the occurring context, then `fun f x = 1` defines variable name `f`.

9. **`fn` expression.** The variables occurring in argument patterns are defined. For example, if `x` is not defined as a data constructor in the occurring context, then `fn x => x` defines variable name `f`.

10. **`case` expression.** The variables occurring in case patterns are defined. For example, if `x` is not defined as a data constructor in the occurring context, then `case y of A x => x` defines variable name `x`.

11. **SQL expression.** SQL expressions that begins with `_sql` may contain variable definitions. They are defined in Chapter 22.

These defined names have their scopes (the extents where the defined names can be referenced). These scopes of defined names are nested according to the inductively defined syntax. As a block-structured language in the Algol family, SML# adopts static scoping rules, under which definition and reference relation is determined at compile time, and new definition hides the old definition of the same name in the syntax introduced by the definition.

In the following, we list the static scopes introduced by scope delimiting syntactic constructs. The actual scope of a name is the parts of the static scope of its defining syntactic construct that exclude the inner static scopes of the same name of the same class.

- Separate compilation unit.

  A separate compilation unit is a single source file ⟨*srcFule*⟩ `.sml`.

  The scope of the names defined in a top-level declaration of ⟨*srcFile*⟩ `.sml` is the rest of the file.

  An interface file reference from a source file also defines names. Without explicit `_interface` declaration in a source file ⟨*srcFile*⟩ `.sml`, the file ⟨*srcFile*⟩ `.sml` in the same directory is implicitly referenced as its interface file. The scope of the names defined by the reference to an interface file ⟨*smiFilePath*⟩ `.smi` from a source file ⟨*srcFile*⟩ `.sml` is the entire source file. The name defined by a reference to an interface file ⟨*smiFilePath*⟩ `.smi` is the set of all names declared in the interface files required by `_require` declarations in ⟨*smiFilePath*⟩ `.smi`.

- Structure construction : `struct` ⟨*decl*⟩ `end`.

  The scope of a name defined in a top-level declaration in ⟨*decl*⟩ is the rest of its declaration occurrence of ⟨*decl*⟩ .

- Local declaration : `local` ⟨$decl_1$⟩ `in` ⟨$decl_2$⟩ `end`.

  The scope of a name defined in ⟨$decl_1$⟩ is the rest of its declaration occurrence in ⟨$decl_1$⟩ and the entire ⟨$decl_2$⟩ . The scope of a name defined in ⟨$decl_2$⟩ is the rest of its declaration occurrence in ⟨$decl_2$⟩ .

- Let expression : `let` ⟨*decl*⟩ `in` ⟨*exp*⟩ `end`.

  The scope of a name defined in ⟨*decl1*⟩ is the rest of its declaration occurrence in ⟨*decl*⟩ and ⟨*exp*⟩.

- Function declaration.

  It has the following syntax.

  `fun` ⟨*id*⟩ ⟨$pat_{1,1}$⟩ $\cdots$ ⟨$pat_{1,n}$⟩ `=` ⟨exp_1⟩
  $\vdots$
  `|` ⟨*id*⟩ ⟨$pat_{m,1}$⟩ $\cdots$ ⟨$pat_{m,n}$⟩ `=` ⟨exp_m⟩

  The scope of the function name ⟨*id*⟩ is the list of ⟨exp_1⟩ to ⟨exp_m⟩. The scope of names in each pattern ⟨$pat_{i,j}$⟩ is the corresponding expression ⟨exp_i⟩.

- Function expression.

  It has the following syntax.

  `fn` ⟨pat_1⟩ `=>` ⟨exp_1⟩ `|` $\cdots$ `|` ⟨pat_n⟩ `=>` ⟨exp_m⟩

  The scope of a name defined in a pattern ⟨pat_i⟩ is the corresponding expression ⟨exp_i⟩.

- Case expression.

  It has the following syntax.

  `case` *exp* `of` ⟨pat_1⟩ `=>` ⟨exp_1⟩ `|` $\cdots$ `|` ⟨pat_n⟩ `=>` ⟨exp_m⟩

  The scope of a name in each pattern ⟨pat_i⟩ is the corresponding expression ⟨exp_i⟩.

Chapter 22

SQL Expressions and Commands

SML# includes database queries compliant to the standard SQL as SML# expressions. The SQL queries and their fragments are first-class citizens; therefore, SQL queries as well as SML# expressions have SML# types and are typechecked under the type system of SML#. In addition, SQL queries can be freely combined with any other SML# constructs as long as their types are consistent. For example, you can construct SQL query fragments as data structures, keep them in variables or function arguments, and make a complete SQL query by combining them at runtime.

22.1 SQL Types

22.1.1 SQL Basic Types

Except for NULL, the following SML# types corespond to SQL basic types:

SML#'s basic types	corresponding SQL types
`int`, `intInf`, `word`	integer types
`bool`	BOOLEAN type of SQL:99 (feature ID T031)
`char`	`CHAR(1)` type
`string`	`TEXT` or `VARVHAR` type
`real`	double-precision floating point types
`real32`	single-precision floating point types

In addition to SML# basic types, the following types are defined for the interoperation with SQL:

SML#'s types	corresponding SQL types
`SQL.numeric`	`NUMERIC` type (decimals of maximum precision)
`SQL.decimal`	`DECIMAL` type (alias of `NUMERIC`)

The concrete correspondence between SML# and SQL types depend on the selection of database engines. See Section 22.8.1 for details of the type correspondence for each database engines.

SQL's NULL corresponds to SML#'s NONE of `option` type. Expressions that may evaluate to NULL, such as references to columns without the NOT NULL constraint, have the `option` type of one of the above basic types.

22.1.2 The Type of SQL Logical Expressions

`SQL.bool3` is the SML# type of the SQL boolean expressions, which consists of comarison and logical operators. This `SQL.bool3` type is intoduced just for the typechecking of SQL queries in SML# and thus it does not correspond to any SQL type.

SML# distinguishes the types of boolean expressions `SQL.bool3` and boolean values `bool`. This is due to the historical confusion of the SQL standards on dealing with truth values. Traditionally, the SQL's truth value is a 3-valued boolean consiting of true, false, and unknown, which are not first-class citizens (they cannot be stored in any table, for example). Therefore, there is no literal denoting truth values. The first-class boolean value has been introduced in SQL99 as an optional feature (feature ID T031). However, the optional feature has been criticized since this optional specification includes serious inconsistencies against the SQL core features. Consequently, almost all of RDBMS vendors have not supported the BOOLEAN type. Even after a few decades since SQL99 had shipped out, no popular RDBMS except for PostgreSQL supports the BOOLEAN type.

To avoid misunderstanding and incompatibility due to the confusion, SML# enforces the type distinction between boolean expressions and values. Thus, boolean literal `true` and `false` cannot be used as boolean expressions, and vice versa. See also Section 22.4.2 for boolean literals, and Section 22.4.6 for boolean expressions.

22.1.3 Types for SQL Tables and Schema

SML#'s record and list types are associated with the structures of SQL's tables, views, and schema. SQL tables and views corresponds to the type of a list of a record whose field names are column names and field types are column types. If a column is not defined with the NOT NULL constraint, the type of the column in SML# is the `option` type of one of the basic types. For example, the structure of

```
CREATE TABLE foo (bar INT, baz TEXT NOT NULL);
```

is represented by the following SML# type:

```
{bar : int option, baz : string} list
```

As seen in the above example, the constraints that means "it is not NULL", such as the NOT NULL and PRIMARY KEY constraints, are represented in SML# types and therefore typechecked at compile-time. Any other SQL constraints are not reflected in SML# types and are checked at run-time when an SQL query is executed in a database server.

SQL schema are represented in a record type, each field of which represents the name and structure of each table. For example, the SQL schema

```
CREATE TABLE employee (id INT PRIMARY KEY, name TEXT NOT NULL,
                       age INT NOT NULL, deptId INT, salary INT);
CREATE TABLE department (deptId INT PRIMARY KEY, name TEXT NOT NULL);
```

is represented as follows in SML#:

```
{
  employee : {id : int, name : string, age : int,
              deptId : int option, salary : int option} list,
  department : {deptId : int, name : string} list
}
```

22.1.4 Types for SQL Queries and Their Fragments

Each category of SQL queries and their subexpressions has a different type in SML#. The following table shows the correspondence between the syntax categories of SQL and SML# types:

Syntax categories	SML# type
SQL value expressions	(τ_1 -> τ_2,w) SQL.exp
SQL commands	(τ,w) SQL.command
SQL queries	(τ,w) SQL.query
SELECT clauses	(τ_1,τ_2,w) SQL.select
FROM clauses	(τ,w) SQL.from
WHERE clauses	(τ,w) SQL.whr
ORDER BY clauses	(τ,w) SQL.orderby
OFFSET clauses	(τ,w) SQL.offset
LIMIT clauses	(τ,w) SQL.limit

where τ is either a table or basic type and w is the type atom that identifies a connection to a database server. The meaning of the above SML# types are the following:

- (τ_1 -> τ_2, w) SQL.exp is the type of SQL value expressions that have type τ_2 under the database connection w and row type τ_1.
- (τ_1,τ_2,w) SQL.select is the type of SELECT clauses that transforms tables of type τ_1 to those of type τ_2 under the database connection w.
- For any other category, (τ,w) SQL.X is the type of constructs of X that have type τ under the database connection w.

22.1.5 Types for SQL Handles

SML# gives database connections and query results the following types:

Types	Descriptions
τ `SQL.server`	a description of a connection to a database of type τ
τ `SQL.conn`	a connection handle to a database of type τ
τ `SQL.cursor`	a cursor to access to a table of type τ
(`$\tau$`,w) `SQL.db`	an instance of a database of type τ

The typical usage of these types are the following:

1. The `_sqlserver` expression generates a connection information of the τ `SQL.server` type (see also Section 22.3).

2. The `SQL.connect` function establishes a connection to the server described in the connection information of the τ `SQL.server` type and returns a connection handle of the τ `SQL.conn` type (see also Section 22.8.1).

3. SQL queries are constructed as polymorphic functions of type `['a. ('τ,'a) SQL.db -> (τ' SQL.cursor,'a) SQL.comm` (see also Section 22.1.6).

4. The `_sql` syntax transforms the SQL query function to an SQL execution function of type τ `SQL.conn ->` τ' `SQL.cursc` (see also Section 22.7).

5. Calling this function by giving it a connection handle of the τ `SQL.conn` type as its argument, the SQL query is sent to the server and evaluated on the server. If the evaluation succeeds, the function returns a cursor of the τ' `SQL.cursor` type to access to the result (see also Section 22.7).

6. The `SQL.fetch` or `SQL.fetchAll` function retrieves records of type τ' from the cursor of the τ' `SQL.cursor` type (see also Section 22.8.2).

22.1.6 SML#'s Policy of Typing SQL Expressions

In SML#, SQL expressions are typed through the correspondence between SML# types and the structures of SQL data and expressions defined above. An SQL query, which operates on tables, has a similar type to an SML# expression that does the same thing as the query for lists of records.

For example, an SQL query

```
SELECT t.name AS employeeName, t.age AS employeeAge
FROM employeeTable AS t
WHERE t.age > 20
```

is written in SML# as follows:

```
val Q = fn db => _sql select #t.name as employeeName, #t.age as employeeAge
                      from #db.employeeTable as t
                      where #t.age > 20
```

where `db` is the bound variable that abstracts the database instance on which the query is evaluated.

By the way, this query computes a table from the `employeeTable` table specified in its FROM clause by filtering its rows according to the condition written in the WHERE clause and transforming the rows by expressions in the SELECT clause. The SML# expression that performs the same thing for lists of records can be written easily as follows:

```
val Q' =
  fn db => List.map
             (fn x => {employeeName = #name (#t x), employeeAge = #age (#t x)})
             (List.filter
                (fn x => #age (#t x) > 20)
                (List.map
                   (fn x => {t = x})
                   (#employeeTable db)))
```

The type of this `Q'` is the following:

```
val Q' : ['a#{employeeTable : 'b list},
          'b#{age : int, name : 'c},
          'c.
          'a -> {employeeAge : 'c, employeeName : int} list]
```

In what follows, we refer to this function `Q'` as a *toy program* of `Q`.

Through this correspondence between SQL queries and SML# expresions, SML# gives the above query `Q` the following type, which represents the fact that the query does the same thing for tables as the SML# expression for lists of records of the similar type:

```
val Q : ['a#{employeeTable : 'b list},
         'b#{age : int, name : 'c},
         'c::{int, ...}, 'd.
         ('a, 'd) SQL.db -> ({employeeAge : 'c, employeeName : int} list, 'd)
SQL.query]
```

The kinded type variables occuring in the type of `Q` represent the following facts:

- `'a` represents the fact that the database on which this query runs must have at least a table named `employeeTable` of type `'b`. Any other tables in the database does not affect the evaluation of the query.
- `'b` means that the `employeeTable` table must have at least two columns: the `age` column of type `int` and the `name` column of type `'c`. The `employeeTable` table may have other columns.
- `'c` indicates that the `name` column may be of an arbitrary basic type (in fact, there is another type variable that the overload kind of `'c` refers to, but it is omitted here).
- `'d` means that this query can be sent to a database server through an arbitrary connection handle (this type variable is used to make sure that a query is not across more than one databases).

As seen in the record-polymorphic type of `Q`, an SQL query is inherently polymorphic with respect to databases. SML# infers the most general polymorphic type of SQL queries with a target database abstract.

Similarly to complete SQL queries, several kinds of SQL query fragments may have the natural correspondence to SML# expressions dealing with lists and records. The typing rules introduced in this chapter are defined based on this correspondence.

22.2 Extended ML Expressions for SQL Queries

As defined in Chapter 19, SML# extends the Standard ML expressions with the following constructs for SQL queries:

⟨*exp*⟩	::=	···	
	|	`_sqlserver` (⟨*appexp*⟩)? `:` ⟨*ty*⟩	SQL servers
	|	`_sql` ⟨*pat*⟩ `=>` ⟨*sqlfn*⟩	SQL execution function
	|	`_sql` ⟨*sql*⟩	SQL query fragments
⟨*atexp*⟩	::=	···	
	|	`_sql (` ⟨*sql*⟩ `)`	SQL query fragments

⟨*sql*⟩ and ⟨*sqlfn*⟩ are given below:

⟨*sql*⟩	::=	⟨*sqlexp*⟩	SQL value expressions
	|	⟨*sqlselect*⟩	SELECT queries
	|	⟨*sqlclause*⟩	SQL query clauses
	|	⟨*sqlcommand*⟩	SQL commands
⟨*sqlfn*⟩	::=	⟨*sqlselect*⟩	SELECT queries
	|	⟨*sqlclause*⟩	SQL query clauses
	|	⟨*sqlcommand*⟩	SQL commands

⟨*sqlexp*⟩ is defined in Section 22.4. ⟨*sqlselect*⟩ and ⟨*sqcommand*⟩ are defined in Section 22.5. ⟨*sqlcommand*⟩ is defined in Section 22.6.

In the expressions that begins with `_sql`, except for the position surrounded by `...(` ⟨*exp*⟩ `)` or `(...` ⟨*exp*⟩ `)` (see Section 22.4, 22.5, and 22.6 for details), the following words are recognized as keywords (this list of keywords is reused from Section 17.2). In the sequel, the following words are refered to as *SQL keywords*.

```
asc all begin by commit cross default delete desc distinct fetch first from group
inner insert into is join limit natural next not null offset only on or order
rollback row rows select set update values where
```

The above syntax has the following limitations:

- ⟨*pat*⟩ following `_sql` must not begin with a left parenthesize "`(`."
- The ⟨*sql*⟩ of `_sql` ⟨*sql*⟩ as expressions (top-level) must begin with an SQL keyword.
- Any `_sql` ⟨*sql*⟩ as expressions (top-level) must occur only at the following positions:
  - the immediate right side of `=` in a `val` and `fun` declarations,
  - the position between `in` and `end` in a `let` expression,
  - the inside of a sequential execution expression `(···;···)`,
  - immediately surrounded by parenthesizes that are not a part of any tuple, and
  - the immediate right side of `=>` in a `fn` expression occurring at one of the above four positions.

22.3 Database Server Description: The _sqlserver Expression

The `_sqlserver` expressions denotes the following two things: the connection information of a database server, and the schema the database server has. When the evaluation result of a `_sqlserver` expression is applied to the `SQL.connect` function (see Section 22.8.1), it tries to connect the database server denoted by the `_sqlserver` expression. If the connection is established and it is confirmed that the database connected contains the schema denoted by the `_sqlserver` expression, the `SQL.connect` function returns a connection handle to the server.

A `_sqlserver` expression consists of the following three items in this order: a kind of database to be connected, a parameter specific to the database kind, and a record type representing the shchma that the database is expected to have. The following is the typical example of a `_sqlserver` expression:

```
_sqlserver SQL.postgresql "host=localhost port=5432"
  : {
      employee : {id : int, name : string, age : int,
                  deptId : int option, salary : int option} list,
      department : {deptId : int, name : string} list
    }
```

This example denotes that the PostgreSQL server listening to localhost:5432 manages the database schema corresponding to the record type. See Section 22.8.1 for details of the database kinds and parameters supported in SML#.

In fact, an arbitrary function application expression ⟨*appexp*⟩ may occur between `_sqlserver` and `:`. Also in the above example, `SQL.postgresql` and `"host=localhost port=5432"` are actually a function and its argument. It is allowed to put here an expression that selects a server among different servers of the same schema at runtime. The expression must be of the `SQL.backend` type.

The type of `_sqlserver` ⟨*appexp*⟩ `:` ⟨*ty*⟩ is ⟨*ty*⟩ `SQL.server`. ⟨*ty*⟩ must be a record type representing a database schema as described in Section 22.1.3.

The following syntax are retained for backward compatibility, but it is discouraged to use them in a new program:

- ⟨*appexp*⟩ may be omitted. If omitted, the PostgreSQL server with the default connection parameter is selected.
- ⟨*appexp*⟩ may be a string literal. If so, the string is interpreted as a connection parameter to a PostgreSQL server.

22.4 SQL Value Expressions

An *SQL value expression* is an expression that occurs in an SQL command and evaluates to a value when the SQL command evaluates. An SML#'s SQL value expression constructs a fragment of the SQL value expression as its value. The SQL value expression constructed is almost literally identical to the SML#'s counterpart and evaluated on the database server. For example, the SML#'s SQL value expression

```
_sql(1 + #employee.salary)
```

is evaluated to the SQL value expression

```
1 + employee.salary
```

as a fragment of an SQL command to be sent to a database server.

If an SQL value expression contains a subexpression that can be evaluated by SML#, the subexpression evaluates to a value and the value is embedded in an SQL command to be sent to a database server. For example,

```
_sql(1 + 2 + #employee.salary)
```

is evaluated to the SQL value expression

```
3 + employee.salary
```

of a fragment of an SQL command.

The following set of expressions ⟨*sqlexp*⟩, which includes a subset of value expressions of the standard SQL, is available in SML#:

- SQL value expressions (top-level)

⟨*sqlexp*⟩	::=	⟨*sqlinfexp*⟩	
	|	`not` ⟨*sqlexp*⟩	SQL's logical nagation
	|	⟨*sqlexp*⟩ `and` ⟨*sqlexp*⟩	SQL's logical conjunction
	|	⟨*sqlexp*⟩ `or` ⟨*sqlexp*⟩	SQL's logical disjunction

- SQL infix expressions

⟨*sqlinfexp*⟩	::=	⟨*sqlcastexp*⟩	
	|	⟨*sqlinfexp*⟩ ⟨*vid*⟩ ⟨*sqlinfexp*⟩	infix expresssions

- SQL type cast expressions

⟨*sqlcastexp*⟩	::=	⟨*sqlappexp*⟩	
	|	`(` ⟨*vid*⟩ `)` ⟨*sqlcastexp*⟩	type cast

- SQL function application expressions

⟨*sqlappexp*⟩	::=	⟨*sqlatexp*⟩	
	|	⟨*vid*⟩ ⟨*sqlatexp*⟩	function applications
	|	⟨*sqlappexp*⟩ ⟨*sqlatexp*⟩	function applications
	|	⟨*sqlappexp*⟩ `is` (`not`)? ⟨*sqlis*⟩	SQL's IS predicates
⟨*sqlis*⟩	::=	`null` | `true` | `false` | `unknown`	

- SQL atomic expressions

⟨*sqlatexp*⟩	::=	⟨*scon*⟩	constant literals
	|	`true`	SML#'s true literal
	|	`false`	SML#'s false literal
	|	`null`	SQL's NULL literal
	|	`#` ⟨*lab*⟩ `.` ⟨*lab*⟩	Reference to a column in a named relation
	|	`#.` ⟨*lab*⟩	Reference to a column in an unnamed relation
	|	⟨*vid*⟩	SML#'s variable reference
	|	`op` ⟨*longvid*⟩	SML#'s variable reference
	|	`(` ⟨*sqlexp*⟩ `,` ⋯, ⟨*sqlexp*⟩ `)`	SML#'s tuples
	|	`((_sql)?` ⟨*sqlselect*⟩ `)`	SQL's SELECT subqueries
	|	(`_sql`)? `exists (`(`_sql`)? ⟨*sqlselect*⟩ `)`	SQL's EXISTS subqueries
	|	`((_sql)?` ⟨*sqlcommand*⟩ `)`	
	|	`((_sql)?` ⟨*sqlclause*⟩ `)`	
	|	`(` ⟨*sqlexp*⟩ `)`	
	|	`( ...` ⟨*exp*⟩ `)`	embedded SQL value expressions

As seen in the definition, some SQL value expressions may begin with the keyword `_sql`. This is allowed just for the inner one of a nested SQL value expressions to be written in the same manner of the outermost one. These `_sql` keywords are simply ignored.

The type of an SQL value expression ⟨*sqlexp*⟩ is determined under the type τ' of a set of tables and w of the database connection identification, both of which are given in the static context. The type of every subexpression in an SQL value expression is given under the same τ' and w. In what follows, we write that the type of an expression e is $(T, w) \triangleright \tau$ if e has type τ under τ' and w. If τ' and w do not need to be described, we simply write that the type of e is τ.

22.4.1 Expressions evaluated by SML#

An SQL value expression satisfying the following inductive condition is *evaluated by SML#* and their values are embedded in the SQL command to be sent to the database server.

1. A constant expression ⟨*scon*⟩ is evaluated by SML#.
2. Variable expressions ⟨*vid*⟩ and `op` ⟨*longvid*⟩ are evaluated by SML#.
3. A tuple `(` ⟨*sqlexp*$_1$⟩ `,` $\cdots$, ⟨*sqlexp*$_n$⟩ `)` is evaluated by SML# if ⟨*sqlexp*$_i$⟩ is evaluated by SML# for any i.
4. A function application ⟨*vid*⟩ ⟨*sqlatexp*⟩ is evaluated by SML# if ⟨*sqlatexp*⟩ is evaluated by SML#.
5. An infix expression ⟨*sqlinfexp*$_1$⟩ ⟨*vid*⟩ ⟨*sqlinfexp*$_2$⟩ is evaluated by SML# if ⟨*sqlinfexp*$_i$⟩ is evaluated by SML# for any i.
6. An SML#'s function application ⟨*sqlappexp*⟩ ⟨*sqlatexp*⟩ is evaluated by SML#. A syntax error occurs if either ⟨*sqlappexp*⟩ or ⟨*sqlatexp*⟩ is not evaluated by SML#.

For example, consider the following SQL query:

```
val q = _sql db => select SOME 1 + sum(#a.b) from #db.a group by #a.c;
```

The subexpression `SOME 1` is evaluated by SML# and its value is embedded in the SQL query fragment (if `SOME` is not defined, it causes an undefined variable error). `sum(#a.b)` is not evaluated by SML# and therefore embedded in the query without any modification. The SQL query that is sent to the server when executing `q` is the following:

```
SELECT 1 + SUM(a.b) AS "1" FROM a GROUP BY a.c
```

The type of expressions that are evalutated by SML# must be one of the SQL basic types defined in Subsection 22.1.1.

Each subexpression evaluated by SML# in an SQL value expression is evaluated immediately at the time when the SQL value expression is evaluated. The evaluation order of such subexpressions are also same as SML# expressions. An SQL value expression is expansive, i.e, avoided to have polymorphic types, if it contains a function application in a subexpression evaluated by SML#. Since SQL queries are polymorphic in almost cases, SQL value expressions including function applicaions evaluated by SML# often cause "value restriction" warnings, as seen in the following example:

```
# _sql(1 + 2);
none:~1.~1-~1.~1 Warning:
(type inference 065) dummy type variable(s) are introduced due to value
restriction in: it
val it = _ : (?X1 -> int, ?X0) SQL.exp
```

To use an SQL expression polymorphically or to put it at the program toplevel, enclose it with `fn () =>` $\cdots$ to avoid the warning. For example, in contrast to the above, the following example does not cause the warning:

```
# fn () => _sql(1 + 2);
val it = fn : ['a, 'b. unit -> ('a -> int, 'b) SQL.exp]
```

22.4.2 SQL constant expressions

Any constant expression in an SQL value expression is evaluated by SML#. The SML#'s syntax is used to write SQL constants; therefore, they looks differently from the standard SQL. In particular, string literals are totally different than the standard one.

Note that neither the `true` nor `false` has `SQL.bool3` type. They denotes a first-class boolean value and therefore they are distinguished from boolean expressions. See Section 22.1.2 for the distinction between boolean expressions and values.

The UNKNOWN literal defined in SQL99 feature ID T031 is not provided in SML#. This is due to the fact that PostgreSQL, the only implementation of the T031 feature, does not have the UNKNOWN litreral, and even if another implementation of T031 would appear in the future, UNKNOWN can be substituted by NULL without changing its meaning (this interchangable use of UNKNOWN and NULL has been criticized since it is not compatible with the SQL core).

22.4.3 SQL identifier expressions

All the ⟨*vid*⟩ and `op` ⟨*longvid*⟩ references to the SML# variables. A ⟨*longvid*⟩ with one or more structure identifiers must begin with the `op` keyword.

In the SQL value expressions, the infixity of some identifiers are declared as follows:

```
infix 7 %
infix 5 like ||
nonfix mod
```

The type and value of an identifier expression is varied depending on whether or not the identifier occurs in an expression evaluated by SML#. If an identifier occurs in an expression evaluted by SML#, its type and value are those of the identifier bound in the current context of the SML# program. Otherwise, the type and value of an identifier are those of the identifier bound in the `SQL.Op` structure.

For example, the two `+` identifiers occured in the expression

```
_sql(1 + 2 + #a.b)
```

have different meanings. The first `+` is the `+` bound in the current SML# environment, and the second `+` is `SQL.Op.+`.

An identifier occuring in an expression that is not evaluated by SML#, except for ⟨*vid*⟩ of `(` ⟨*vid*⟩ `)` ⟨*sqlappexp*⟩ 22.4.5), is embedded in the SQL query after translating it by the following rules:

- All lower case alphabets are replaced with upper case alphabets.

The value of an identifier defined in the `SQL.Op` structure is referred only when the toy program of the SQL query is executed (see Subsection 22.8.3). Its type is used for typing SQL value expressions. For example, the type of `_sql(1 + #a.b)` in SML# can be determined by looking for the type of `SQL.Op.+`.

22.4.4 SQL function applications and infix expressions

SQL function application and infix expressions are parsed in the same way as those of SML#, except for builtin logical expressions. The associatibity of infix expressions is decided by SML#'s infix declarations. Thus, SML#'s SQL infix operators may have different associatibity than the standard SQL in accordance with the use of the infix declarations.

Unlike the standard SQL, the syntax of SML#'s function application expressions does not require the function arguments to be parenthesized, but requires at least one delimiter or space between the function and arguemnts. However, particularly in an SQL expressions, it is suggested to use parenthesizes for the arguments and omit the space between the function and arguments so that it looks much closer to the standard SQL. This is completely allowed in the grammar of Standard ML. For example, an aggregate function can be used in a SQL query like as follows:

```
fn db => _sql select avg(#e.age)
              from #db.employee as e
              group by ()
```

The subexpression `avg(#e.age)` is a function application expression that applies the `SQL.Op.avg` function to the argument `#e.age`.

Each of the SQL infix operators and functions, except for builtin logical operators, is provided as an SML# library function defined in the `SQL.Op` structure. The list of SQL functions and operators available in expressions that are not evaluated by SML# is shown in Section 22.9.2.

An SQL function application expression and SQL infix expression is typechecked in the same way as those of SML#.

22.4.5 Type cast expressions

In a SML#'s SQL value expression, the expression consisting of a single identifier surrounded by parenthesizes has a special meaning. By putting an identifier surrounded by parenthesizes in front of an expression, the value of the expression is applied to the function identified by the identifier. The identifier and its surrounding parenthesizes do not appear in the SQL query to be sent to the database server. An identifier surrounded by parenthesizes must be defined in the `SQL.Op` structure. In what follows, an expression prefixed with an identifier surrounded by parenthesizes is referred to as *a type cast expression*.

A type cast expression corresponds to implicit type cast in SQL, which is not supported by SML#'s type system. One of its typical usage is to deal with the `option` and numeric types. As described in Section 22.1, SML# distinguish values being possibly NULL from non-NULL values by using `option` type. In contrast, the standard SQL allows NULL in any type. Due to this difference, the SML# compiler reports a type error even for a query type-correct in the standard SQL. Similarly, SML# does not allow implicit numeric type conversion, which the standard SQL allows. For example, consider the following that queries people younger than the average in each deparment:

```
fn db => _sql select #e.department as department, #e.name as name
              from #db.empoloyee as e
              where #e.age < (select avg(#t.age)
                              from #db.employee as t
                              where #t.department = #e.department
                              group by ())
```

This expression is typechecked, but the type of `age` column of `employee` table is infered as `SQL.numeric option`. Thus, this query cannot be executed if the `age` column is of `int`. The source of this inference is that the result of the aggregate function `avg` is compared with `#e.age`. The result type of `avg` is `SQL.numeric option`. The comparison operator forces the two expressions to have the same type. Therefore, the type of `#e.age` is `SQL.numeric option`. In the standard SQL, `age` may have an arbitrary numeric type since its type is implicitly casted to `NUMERIC`.

For the above example, inserting `Num` function to the left side of the comparison operator allows `age` to have an arbitrary numeric type.

```
fn db => _sql select #e.department as department, #e.name as name
              from #db.empoloyee as e
              where (Num)#e.age < (select avg(#t.age)
                                   from #db.employee as t
                                   where #t.department = #e.department
                                   group by ())
```

This `(Num)` is ignored when an SQL query is composed; therefore, regardless of the existence of `(Num)`, the SQL query to be sent to the database server is not changed. The `(Num)` is evaluated statically only for the typechecking of the SQL value expression.

For the set of identifiers provided for type cast expressions, see Subsection 22.9.1.

22.4.6 SQL logical expressions

The following logical operators are built in the syntax of expressions:

- `not` ⟨*sqlexp*⟩
- ⟨*sqlexp*$_1$⟩ `and` ⟨*sqlexp*$_2$⟩
- ⟨*sqlexp*$_1$⟩ `or` ⟨*sqlexp*$_2$⟩

- ⟨*sqlexp*⟩ `is` (`not`)? ⟨*sqlis*⟩

The `and` operator has an additional restriction on its syntax: to avoid conflict with other SML# syntax, `and` may occur only if it is parenthesized. For example, the following causes a parse error:

```
# fn () => _sql where #t.c >= 10 and #t.c <= 20;
(interactive):1.31-1.35 Error: Syntax error: deleting AND HASH
```

By surrounding the expression including the `and` operator with parenthesizes, the parse error is avoided, as seen in the following:

```
# fn () => _sql where (#t.c >= 10 and #t.c <= 20);
val it = fn : ['a#t: 'b, 'b#c: int, 'c.
               unit -> ('a list -> 'a list, 'c) SQL.whr]
```

The type of a logical expression is `SQL.bool3` if its all subexpressions is of the `SQL.bool3` type. Similarly, all SQL comparison operators provided as SML# library functions return a query of the `SQL.bool3` type.

As described in Section 22.1.2, to avoid confusion on dealing with boolean values in the standard SQL, SML# distingishes the types of boolean expressions and boolean values. Note that boolean value expressions, such as a reference to a column of the BOOLEAN type and the boolean value literals like `true`, is not allowed to be a boolean expression. The following example causes a type error:

```
# fn () => _sql(true is false);
(interactive):1.14-1.26 Error:
(type inference 016) operator and operand don't agree
operator domain: SQL.bool3
operand: 'HBP::bool
```

In contrast, the following does not cause type error

```
# fn () => _sql(#t.c = true is false);
val it = fn : ['a, 'b. unit -> ('a -> SQL.bool3, 'b) SQL.exp]
```

because `#t.c = true` is an boolean expression whereas `true` itself denotes an boolean value.

22.4.7 SQL column reference expressions

To reference a certain column of a certain table, write `#` ⟨lab_1⟩ `.` ⟨lab_2⟩, where ⟨lab_1⟩ and ⟨lab_2⟩ are the names of the table and column, respectively. The type of a column reference is that of the column ⟨lab_2⟩ of the table ⟨lab_1⟩ as specified in the type of table sets given by the static context.

Unlike the standard SQL, which sometimes allows to omit the table name in a column reference, SML# requires the table name explicit in all the column reference expressions. In addition, column references must be prefixed with `#` to avoid the conflict with other syntax.

Also unlike the standard SQL, table names and column names in SML# are case-sensitive. For example,

```
SELECT Employee.name FROM EMPLOYEE
```

is a valid query in the standard SQL, but in SML#,

```
_sql db => select #Employee.name from #db.EMPLOYEE
```

causes an type error because the table name referenced is not found. In the SQL value expression to be sent to a server, the table and column names appears in the same case as the SML#'s counterpart.

Sometimes in SQL, a column in an anonymous table is referenced. A typical example of this situation is the ORDER BY clause that refers to a column computed by the SELECT clause. In SML#, such kind of column references is written as `#.` ⟨*lab*⟩. Its type is the type of the column ⟨*lab*⟩. For example, the SQL query

```
SELECT e.name AS name, e.age + 1 AS nextAge
FROM employee AS e
ORDER BY nextAge
```

is written in SML# as follows:

```
fn db => _sql select #e.name as name, #e.age + 1 as nextAge
              from employee as e
              order by #.nextAge
```

22.4.8 SQL Subqueries

The following two kinds of subqueries are allowed in SML#:

- SELECT subqueries: `(` ⟨*sqlselect*⟩ `)`
- EXIST subqueries: `exists (` ⟨*sqlselect*⟩ `)`

An additional `_sql` keyword may appear just before the first keyword of a subquery so that it looks like other SML#'s SQL expressions. These `_sql`s are simply ignored.

A SELECT subquery `(` ⟨*sqlselect*⟩ `)` must return a 1-column and 1-row result. The type system of SML# enforces the 1-column property of a subquery and therefore it is checked at compile time. The type of ⟨*sqlselect*⟩ must be

`({1:` τ`} list,` w`) SQL.query`

in SML#. If it is satisfied, the type of the subquery expression `(` ⟨*sqlselect*⟩ `)` is $(T, w) \triangleright \tau$ for some τ'. The number of rows in the result of the subquery is checked at runtime by a database server. If the subquery returns zero or more than one rows, the `SQL.Exec` exception is raised.

For an EXIST subquery `exists (` ⟨*sqlselect*⟩ `)`, if ⟨*sqlselect*⟩ is of type

`(`τ `list,` w`) SQL.query`,

then the entire expression is of type $(\tau', w) \triangleright$ `SQL.bool3` for some τ'.

22.4.9 Embedded SQL value expressions

To obtain a first-class SML# object of an SQL value expression ⟨*sqlexp*⟩, write an expression `_sql(` ⟨*sqlexp*⟩ `)`. The type of `_sql(` ⟨*sqlexp*⟩ `)` is `(`τ' `->` τ`,` w`) SQL.exp` if ⟨*sqlexp*⟩ is of type $(\tau', w) \triangleright \tau$. For example, the type of

```
val q = _sql(1 + #employee.salary)
```

is

```
val q : [ 'a#{employee: 'b}, 'b#{salary: int}, 'w. ('a -> int, 'w) SQL.exp ].
```

To embed a fragment of SQL value expression obtained in such a way in another SQL value expression, use the `(...` ⟨*exp*⟩ `)` expression. For example, the following expression is constructed by embedding `q` in it:

```
_sql((...q) > 10)
```

This expression denotes the following SQL query:

```
1 + employee.salary > 10
```

The type of `(...` ⟨*exp*⟩ `)` is $(\tau', w) \triangleright \tau$ if the type of ⟨*exp*⟩ is `(`τ' `->` τ`,` w`) SQL.exp`. If τ' is inconsistent with the type of tables referenced by the expression where the ⟨*exp*⟩ is embedded in, a type error occurs. For example, the following expression causes a type error:

```
_sql((...q) > 10 and #employee.salary = "abc")
```

The ⟨*exp*⟩ in the `(...` ⟨*exp*⟩ `)` notation is not an SQL value expression but an SML# expression. In ⟨*exp*⟩, SQL keywords are not keywords but ordinary identifiers as well as ordinary SML# expressions.

Although an embedded SQL value expresssion constitutes a constant expression in the resulting SQL value expression, the constant expression is not evaluated by SML#. Whether or not an SQL value expression is evaluated by SML# is determined only by the static syntacitic structure of the expression. For example, the following function `nat` n does not return the value n, but returns the SQL value expression `1 + 1 +` $\cdots$ `+ 1` where the number of `1`s is n

```
# fun nat 1 = _sql(1)
    | nat n = _sql(1 + (...nat (n - 1)));
val nat = fn : ['a, 'b. int -> ('a -> int, 'd) SQL.exp]
# SQL.expToString (nat 5);
val it = "(1 + (1 + (1 + (1 + 1))))" : string
```

22.5 SELECT queries

SML# allows to construct a SELECT query by defining each of its clauses separately and composing the clauses into one query. The clauses ⟨*sqlclause*⟩ of which a SELECT query consists is given below:

⟨*sqlclause*⟩	::=	⟨*sqlSelectClause*⟩	SELECT clauses
	|	⟨*sqlFromClause*⟩	FROM clauses
	|	⟨*sqlWhereClause*⟩	WHERE clauses
	|	⟨*sqlOrderClause*⟩	ORDER BY clauses
	|	⟨*sqlOffsetClause*⟩	OFFSET clauses
	|	⟨*sqlLimitClause*⟩	LIMIT clauses

The details of each clause is shown below in a subsection of this section.

A SELECT query is constructed by combining these clauses and the GROUP BY clause ⟨*sqlGroupClause*⟩. The syntax of the SELECT queries is the following:

⟨*sqlselect*⟩	::=	⟨*sqlSelectClause*⟩	SELECT query with SELECT clause
		⟨*sqlFromClause*⟩	
		(⟨*sqlWhereClause*⟩)?	
		(⟨*sqlGroupClause*⟩)?	
		(⟨*sqlOrderClause*⟩)?	
		(⟨*sqlLimitClause*⟩)?	
	|	`select ... (` ⟨*exp*⟩ `)`	SELECT query whose SELECT clause will be embedded
		⟨*sqlFromClause*⟩	
		(⟨*sqlWhereClause*⟩)?	
		(⟨*sqlGroupClause*⟩)?	
		(⟨*sqlOrderClause*⟩)?	
		(⟨*sqlLimitClause*⟩)?	
	|	`select ... (` ⟨*exp*⟩ `)`	embedded SELECT queries

The first two syntaxes constructs a SELECT query. A SELECT query must contain a SELECT clause and FROM clause. Other clauses are optional. While some RDBMSes accepts SELECT queries without FROM clauses, SML# does not allow such queries. The meaning of `select ... (`⟨*exp*⟩`)` in the second form is decribed later.

A SELECT query has type `(`τ`, `w`) SQL.query` if the following conditions are met:

- ⟨*sqlFromClause*⟩ has type `(`τ_1`, `w`) SQL.from` for some τ_1.
- If ⟨*sqlWhereClause*⟩ exists, it must have type `(`τ_1 `->` τ_1`, `w`) SQL.whr`.
- If a GROUP BY clause exists, a row group type τ_2 is calculated from τ_1 (see Section 22.5.4 for details). Otherwise, $\tau_2 = \tau_1$.
- ⟨*sqlSelectClause*⟩ must have type `(`τ_2`, `τ`, `w`) SQL.select`.
- If ⟨*sqlOrderClause*⟩ exists, it must have type `(`τ `->` τ`, `w`) SQL.orderby`.
- If ⟨*sqlLimitClause*⟩ exists, it must have type `(`τ `->` τ`, `w`) SQL.limit`.

The following is an example of a complete SELECT query:

```
val q = fn db => _sql select #t.name as name, #t.age as age
                      from #db.employee as t
                      where #t.age >= 20
```

The third form `select...(`⟨*exp*⟩`)` embeds the result of the SML# expression ⟨*exp*⟩ as a SELECT query. The type of ⟨*exp*⟩ must be `SQL.query`. For example, the following code embeds the query `q` in another query `q2` as a subquery:

```
val q2 = fn db => _sql select #t.name as name
                       from (select...(q db)) as t
                       where #t.name like "%Taro%"
```

As seen in the second form of the SELECT query and the syntax of each clause shown in the subsections, except for some particular clauses, it is allowed to write the clause name followed by `...(`⟨*exp*⟩`)` to embed the result of ⟨*exp*⟩ as a clause of the clause name. By using this nontation, the above example of a SELECT query can be decomposed into a series of `val` definitions as follows:

```
val s = _sql select #t.name as name, #t.age as age
val f = fn db => _sql from #db.employee as t
val q = fn db => _sql select...(s) from...(f db)
```

The ⟨*exp*⟩ in the `...(`⟨*exp*⟩`)` notation is not an SQL value expression but an SML# expression. In ⟨*exp*⟩, SQL keywords are not keywords but ordinary identifiers as well as ordinary SML# expressions.

22.5.1 SELECT clauses

The syntax of SELECT clauses ⟨*sqlSelectClause*⟩ is the following:

⟨*sqlSelectClause*⟩ ::= `select` (⟨*distinct_all*⟩)? ⟨*sqlSelectField*⟩ `,` ⋯, ⟨*sqlSelectField*⟩
⟨*distinct_all*⟩ ::= `distinct` | `all`
⟨*sqlSelectField*⟩ ::= ⟨*sqlexp*⟩ (`as` ⟨*lab*⟩)?

A SELECT clause consists of one or more fields of the form ⟨*sqlSelectField*⟩. Each field has a label ⟨*lab*⟩. If the `as` ⟨lab_i⟩ of the i-th field (the first field is 1st) is omitted, it is complemented by adding `as` i. No two fields may have the same label.

For ⟨*sqlSelectClause*⟩ with n fields, if the ⟨$sqlexp_i$⟩ of its i-th field ($1 \leq i \leq n$) ⟨$sqlexp_i$⟩ `as` ⟨lab_i⟩ has the $(\tau, w) \triangleright \tau_i$ type, the type of entire ⟨*sqlSelectClause*⟩ is of the type

`(`τ`, {`⟨lab_1⟩ `:` τ_1`, ...,` ⟨lab_n⟩ `:` τ_n`} list,` w`) SQL.query`.

22.5.2 FROM clauses

The syntax of the From clauses ⟨*sqlFromClause*⟩ is the following:

⟨*sqlFromClause*⟩ ::= `from` ⟨*sqlTable*⟩ `,` ⋯ `,` ⟨*sqlTable*⟩
| `from ... (`⟨*exp*⟩`)`

The first syntax constructs a FROM clause. The second form embeds the value of ⟨*exp*⟩ in the ontext as a FROM clause. ⟨*exp*⟩ must be of the `SQL.from` type.

The commas in the first form sometimes conflicts with those of SML#'s tuple expressions. For example,

```
(1, _sql from #db.t1, #db.t2, #db.t3)
```

is ambiguous because there are two possible interpretations: this denotes a pair of an integer and FROM clause, or a 4-tuple whose second element is a FROM clause. The FROM clause must be parenthesized as either

```
(1, _sql (from #db.t1, #db.t2, #db.t3))
```

if the expression means the former, or

```
(1, _sql (from #db.t1), #db.t2, #db.t3)
```

if it means the latter. This is why the syntactic restriction described in Section 22.2 is introduced. In contrast to the above example,

```
(_sql from #db.t1, #db.t2, #db.t3)
```

is a single FROM clause expression with no ambiguity because it begins with `_sql`.

A table expression ⟨*sqlTable*⟩ denotes a table. Its syntax is as follows:

⟨*sqlTable*⟩	::=	`#`⟨*vid*⟩`.`⟨*lab*⟩	reference to a table
	|	⟨*sqlTable*⟩ `as` ⟨*lab*⟩	labeling expressions
	|	`(` (`_sql`)? ⟨*sqlselect*⟩ `)`	table subqueries
	|	⟨*sqlTable*⟩ (`inner`)? `join` ⟨*sqlTable*⟩ `on` ⟨*sqlexp*⟩	inner join of two tables
	|	⟨*sqlTable*⟩ `cross join` ⟨*sqlTable*⟩	product of two tables
	|	⟨*sqlTable*⟩ `natural join` ⟨*sqlTable*⟩	natural join of two tables
	|	`(`⟨*sqlTable*⟩`)`	

The labeling expressions must has the following rules:

- The labeling expressions has the strongest associativity among the table expressions.
- If a table reference `#`⟨*vid*⟩`.`⟨*lab*⟩ is not labeled with a labeling expression, it is labeled by the same name as the table as if `as` ⟨*lab*⟩ is specified.

The syntax of ⟨*sqlTable*⟩ is restricted by the following rules:

- All `as` ⟨*lab*⟩s occurring in a ⟨*sqlFromClauses*⟩ must be distinct. Thus, if a table is referenced twice or more times in a FROM clause, the references of the table must be labeled differently by `as`.
- Each of the ⟨$sqlTable_1$⟩ and ⟨$sqlTable_2$⟩ in a ⟨$sqlTable_1$⟩ `natural join` ⟨$sqlTable_2$⟩ must be neither an inner or cross join.
- The ⟨*sqlTable*⟩ of a ⟨*sqlTable*⟩ `as` ⟨*lab*⟩ must be neither an inner or cross join.

For a a FROM clause with n labels of ⟨$sqlTable_i$⟩ `as` ⟨lab_i⟩ ($1 \leq i \leq n$), if the type of each ⟨$sqlTable_i$⟩ is τ_i, the type of entire FROM clause is the following:

`({`⟨lab_1⟩ `:` τ_1`, ...,` ⟨lab_n⟩ `:` τ_n`} list,` w`) SQL.from`

The type of ⟨$sqlTable_i$⟩ is decided as follows:

- The type of `#`⟨*vid*⟩ `.` ⟨*lab*⟩ is τ' where the type of ⟨*vid*⟩ is `(`τ`,` w`) SQL.db` and τ is a record type including at least a ⟨*lab*⟩ `:`τ' field.
- The type of ⟨*sqlTable*⟩ `as` ⟨*lab*⟩ is identical to that of ⟨*sqlTable*⟩.
- The type of `(` ⟨*sqlselect*⟩ `)` is τ where ⟨*sqlselect*⟩ has type `(`τ`,`w`) SQL.query`.
- The type of ⟨$sqlTable_1$⟩ `natural join` ⟨$sqlTable_2$⟩ is the record type obtained by performing the natural join operation on the record types of the two tables.

SML# does not compute the type of inner and cross joins. Therefore, inner and cross joins cannot be labeled by `as` directly (syntactically restricted). The type of these joins are represented by the record type of the entire FROM clause.

A FROM clause computes a table obtained by joining the tables it refers to. The record type `{`⟨lab_1⟩ `:` τ_1`, ...,` ⟨lab_n⟩ `:` τ_n`}` of a FROM clause represents a row of the computed table. Each field of the record type corresponds to a labeled subset of columns. The SQL column reference expressions occurring in other clauses refer to these labels.

22.5.3 WHERE clauses

Here is the syntax of the WHERE clauses ⟨*sqlWhereClause*⟩:

⟨*sqlWhereClause*⟩ ::= `where` ⟨*sqlexp*⟩
| `where ... (` ⟨*exp*⟩ `)`

The first form construct a WHERE clause. Its type is `(`τ `->` τ`,` w`) SQL.whr` if ⟨*sqlexp*⟩ has type $(\tau, w) \triangleright$ `SQL.bool3`.

The second form embeds the result of ⟨*exp*⟩ in the current context. ⟨*exp*⟩ must be of the `SQL.whr` type.

22.5.4 GROUP BY clauses

The syntax of the GROUP BY clauses ⟨*sqlGroupClause*⟩ is the following:

⟨*sqlGroupClause*⟩ ::= `group by` ⟨*sqlexp*⟩ `,` $\cdots$ `,` ⟨*sqlexp*⟩ (`having` ⟨*sqlexp*⟩)?
| `group by ()`

Different from other clauses, the `...(`⟨*exp*⟩`)` notation is not allowed for the GROUP BY clauses and therefore they cannot be separated from the SELECT queries. Syntactically, a GROUP BY clause must occur along with a SELECT and FROM clauses.

Each ⟨$sqlexp_i$⟩ comma-separated in a GROUP BY clause must be of type $(\tau, w) \triangleright \tau_i$ where the type of the associated FROM clause is `(`τ`,` w`) SQL.from`. The GROUP BY clause splits the table computed by the FROM clause to the groups of rows by using keys specified in ⟨*sqlexp*⟩s, as described later.

A GROUP BY clause may have just one HAVING clause. The value expression of a HAVING clause is a condition to filter the row groups; therefore, its type must be $(\tau', w) \triangleright$ `SQL.bool3`.

The second form `group by ()` is one of the standard SQL syntax that makes a single group of all rows. In a conventional SQL, if an aggregate function is used without a GROUP BY clause, the query aggregates the entire table implicitly. For example, the SQL query

```
SELECT avg(e.age) FROM employee AS e
```

is a correct SQL query that computes the average of the `e.age` column. In contrast, SML# forces such an aggregating query to have `group by ()`. The above query must be written in SML# as follows:

```
fn db => _sql select avg(#e.age) from #db.employee as e group by ()
```

This SML# expression is evaluated to the SQL query

```
SELECT avg(e.age) FROM employee AS e GROUP BY ()
```

as in the SML# expression.

The type τ' of the row groups computed by a GROUP BY clause is computed roughly in the following steps:

1. Let the type of rows before grouping be

   $$\tau = \{k_1 : \tau_1,\ \ldots,\ k_n : \tau_n\}$$

   where the type of the FROM clause is τ `list`.

2. Group the rows in lists. Hence, the group type is naturally τ `list list`.

3. Transpose the row group type $\{k_1 : \tau_1,\ \ldots,\ k_n : \tau_n\}$ `list` to $\tau^t = \{k_1 : \tau_1$ `list,` $\ldots,\ k_n : \tau_n$ `list}`.

For the transposition in the step 3, the set of columns $k_1, \ldots, k_n$ must be determined at compile time, as opposed to the record-polymorphic property of queries. SML# obtains the set of columns from column reference expressions ocurring in the entire query with a GROUP BY clause. For each column reference `#`$\langle lab_1 \rangle$ `.`$\langle lab_2 \rangle$ occurring as either a key specified in the GROUP BY clause or column reference expression referring to a grouped value, if the column reference is identical to one of the group keys, the column refers to the single key value and therefore the type of the column is not changed in the result of the GROUP BY clause. Otherwise, the type of the column is the list type of its original type. Any other columns, which are never refered to in the query, are ignored and omitted in the result type of the GROUP BY clause.

Note that this type computation is performed based on the syntactic context and hence does not consider variable references. For example,

```
val q = fn db => _sql select #e.department, avg(#e.salary)
                      from #db.employee as e
                      group by #e.department
```

is a typechecked query with a GROUP BY clause. However,

```
val s = _sql select #e.department, avg(#e.salary)
val q = fn db => _sql select...(s)
                      from #db.employee as e
                      group by #e.department
```

causes a type error because the second example does not have any SELECT clause construction but an embedding, in contrast to the first example that have both a SELECT and GROUP BY clauses in the same syntax of a query. In the first example, the type of the row groups has `#e.department` and `#e.salary` columns as expected, but in the second example, the row groups is regarded to have no column since no column reference occur in the same query syntax. As a good manner, when you write a query with `group by`, you should not use the `...(`$\langle exp \rangle$`)` notation for its SELECT clause.

Notes for the observant readers: even when the SELECT clause is sepearated from a query with GROUP BY, if the embedded SELECT clause only refers to the group keys, the query does not cause type errors. For example, edit the above example by removing `avg(#e.salary)` from the definition of `s` as follows:

```
val s = _sql select #e.department
val q = fn db => _sql select...(s)
                      from #db.employee as e
                      group by #e.department
```

This program does not cause type errors because `#e.department` is a group key and therefore it is included in the result type of GROUP BY.

22.5.5 ORDER BY clauses

The syntax of the ORDER BY clauses ⟨*sqlOrderclause*⟩ is the following:

⟨*sqlOrderClause*⟩ ::= `order by` ⟨*sqlOrderKey*⟩ `,` ⋯ ⟨*sqlOrderKey*⟩
| `order by ... (` ⟨*exp*⟩ `)`
⟨*sqlOrderKey*⟩ ::= ⟨*sqlexp*⟩ (⟨*asc_desc*⟩)?
⟨*asc_desc*⟩ ::= `asc` | `desc`

The first form constructs an ORDER BY clause. The second form embeds the result of ⟨*exp*⟩ in the current context as an ORDER BY clause. ⟨*exp*⟩ must be of the `SQL.orderby` type.

An ORDER BY clause rearranges the rows computed by the SELECT clause. For the SELECT query computing the rows of type τ, Each ⟨$sqlexp_i$⟩ in the ORDER BY clause must be $(\tau, w) \triangleright \tau_i$, and the entire ORDER BY clause must be of type `(`τ `->` τ`, `w`) SQL.orderby`.

There are several variations of ORDER BY clause in SQL since it has been extended by the database vendors and standard trucks without backward compatibility. The convention SML# adopts is that an ORDER BY clause may refer to only the result of the SELECT clause. The column reference expression with anonymous table `#.` ⟨*lab*⟩ is used to refer to a column of the SELECT clause from the ORDER BY clause. The following shows an example:

```
fn db => _sql select #e.name as name, #.age as age
              from #db.employee as e
              order by #.age
```

While many database engines allows for ORDER BY to refer to columns other than those of SELECT, SML# prohibits such references. The following program causes type errors, for example:

```
fn db => _sql select #e.name as name, #.age as age
              from #db.employee as e
              order by #e.department
```

22.5.6 OFFSET or LIMIT clauses

The OFFSET and LIMIT clauses have same meaning; they returns the rows only in the given range. The major difference of these clauses is who specify them: the OFFSET clause is specified in the standard SQL, and the LIMIT clause is a widely-accepted vender extention. SML# supports both.

The syntax of them ⟨*sqlOffsetOrLimitClause*⟩ is as follows:

⟨*sqlOffsetOrLimitClause*⟩ ::= `sqlOffsetClause` | `sqlLimitClause`
⟨*sqlOffsetClause*⟩ ::= `offset` ⟨*sqlatexp*⟩ ⟨*row˙rows*⟩ (⟨*sqlFetchClause*⟩)?
⟨*sqlFetchClause*⟩ ::= `fetch` ⟨*first˙next*⟩ ⟨*sqlatexp*⟩ ⟨*row˙rows*⟩ `only`
⟨*row_rows*⟩ ::= `row` | `rows`
⟨*first_next*⟩ ::= `first` | `next`
⟨*sqlLimitClause*⟩ ::= `limit` ⟨*sqlexp*⟩ (⟨*sqlLimitOffsetClause*⟩)?
| `limit all` (⟨*sqlLimitOffsetClause*⟩)?
⟨*sqlLimitOffsetClause*⟩ ::= `offset` ⟨*sqlexp*⟩

Note that the keyword `offset` is used in two ways in the above syntax: `offset` is either the subclause for a LIMIT clause or the main clause of a OFFSET clause that may have a FETCH subclause. These subclauses are not interchangable. Note also that if an OFFSET clause or its FETCH subclause has a non-constant expression, the expression must be parenthesized.

The type of ⟨*sqlexp*⟩ and ⟨*sqlatexp*⟩ must be $(\{\}, w) \triangleright$ `int`. Therefore, no column reference may appear in these clauses.

Several database engines allows these clauses and subclauses to appear more than one times in a qeury in any order. SML# follows the standard SQL and hence a main clause must be followed by its subclause. A query may contain only one of these clauses.

22.5.7 Corelated Subqueries

If more than one SELECT queries are nested, the inner query is a subquery of the outer query. A subquery is referred to as a *corelated subquery* if it refers to columns introduced by the FROM clause of its outer query. The following is an example of a corelated subquery in the standard SQL:

```
SELECT e.department AS department, e.name AS name
FROM empoloyee AS e
WHERE e.salary > (SELECT avg(#t.salary)
                  FROM employee as t
                  WHERE t.department = e.department
                  GROUP BY ())
```

In SML#, a subquery may appear as an value expression ⟨*sqlexp*⟩ (see also Section 22.4.8) and in a FROM clause (Section 22.5.2). A subquery may be a corelated subquery if the following conditions are met:

1. If a subquery itself and all of its outer queries have FROM clauses not of the form `from...(` ⟨*exp*⟩ `)`, the subquery may be a corelated subquery.

This is a translation of the above SQL query into SML#.

```
fn db => _sql select #e.department as department, #e.name as name
              from #db.empoloyee as e
              where #e.salary > (select avg(#t.salary)
                                 from #db.employee as t
                                 where #t.department = #e.department
                                 group by ())
```

If one of the nested query has `from...(` ⟨*exp*⟩ `)`, subqueries are not to be corelated. For example, if the above example is editted as follows by replacing its FROM clause with `from...(` ⟨*exp*⟩ `)`

```
let
  val f = fn db => _sql from #db.employee as t
in
  fn db => _sql select #e.department as department, #e.name as name
                from #db.empoloyee as e
                where #e.salary > (select avg(#t.salary)
                                   from...(f db)
                                   where #t.department = #e.department
                                   group by ())
end
```

the `e` of `#e.department` in the subquery is not interpreted as the `e` of `from #db.employee as e` in the outer query, but `e` to be introduced by `from ...(x db)`. Therefore, `x db` must have `e`. Concequently, this example causes a type error. Also in the case when the outer query has `from...(` ⟨*exp*⟩ `)`, such as

```
let
  val f = fn db => _sql from #db.empoloyee as e
in
  fn db => _sql select #e.department as department, #e.name as name
                from...(f db)
                where #e.salary > (select avg(#t.salary)
                                   from #db.employee as t
                                   where #t.department = #e.department
                                   group by ())
```

the subquery is not regarded as corelated one and therefore the `e` of `#e.department` is undefined.

Corelated subqueries and GROUP BY clauses can be combined as expected. For example, the following queries the youngest person who have incomes greater than the average saraly of his/her department:

```
fn db => _sql
  select #e.department, (select min(#t.age)
                         from #db.employee as t
                         where (#t.department = #e.department
                                and (Some) #t.salary > min(#e.salary))
                         group by ())
  from #db.employee as e
  group by #e.department
```

The inner query aggregates the grouped column `#e.salary` of the outer query by the `min` aggregate function. As described in Section `#e.salary`, the type of GROUP BY is computed from the syntactic context. This is also true even if a query has corelated subqueries.

Notes for the observant readers: A corelated subquery must occur in a outer query, but the subquery is not tightly corelated to the outer query similarly to the static scoping of ML variables. The syntactic restriction of corelated subqueries are introduced just to make the type computation of subqueries possible in a static manner. As long as any SQL value expressions including corelated subqueries are first-class citizens, it is possible to detach a corelated subquery A from a nested query B and embed A to another nested query C by exploiting SML# features. In the resulting query, A embedded in C refers to C's tables, not to B. Regardless of the query construction operations, if the type of those operations are consistent, then the resulting SQL query is correct.

22.6 SQL commands

SML# accept the following subset of SQL commands ⟨*sqlcommand*⟩ as well as SELECT queries:

⟨*sqlcommand*⟩	::=	⟨*sqlinsert*⟩	INSERT commands
	|	⟨*sqlupdate*⟩	UPDATE commands
	|	⟨*sqldelete*⟩	DELETE commands
	|	⟨*sqltransaction*⟩	transaction management commands
	|	`(` ⟨*sqlfn*⟩ `;` ⋯ `;` ⟨*sqlfn*⟩ `)`	command sequence
	|	`...` `(` ⟨*exp*⟩ `)`	embedded SQL commands

The type of INSERT, UPDATE, DELETE, and transaction commands is

```
(unit, w) SQL.command
```

Details of these commands are shown in the following subsections.

`(` ⟨$sqlfn_1$⟩ `;` ⋯ `;` ⟨$sqlfn_n$⟩ `)` (n must be greater than 1) is a command consisting of a sequence of commands separated by semicolons. Its type is equal to the type of the last command ⟨$sqlfn_n$⟩.

`...(` ⟨*exp*⟩ `)` embeds the result of ⟨*exp*⟩ as a command.

22.6.1 INSERT commands

The syntax of the INSERT command is the following:

⟨*sqlinsert*⟩	::=	`insert into #` ⟨*vid*⟩ `.` ⟨*lab*⟩ `(` ⟨*lab*⟩ `,` ⋯ `,` ⟨*lab*⟩ `)`
		`values` ⟨*insertRow*⟩ `,` ⋯ `,` ⟨*insertRow*⟩
	|	`insert into #` ⟨*vid*⟩ `.` ⟨*lab*⟩ `(` ⟨*lab*⟩ `,` ⋯ `,` ⟨*lab*⟩ `)`
		`values` ⟨*insertVar*⟩
	|	`insert into #` ⟨*vid*⟩ `.` ⟨*lab*⟩ `((` ⟨*lab*⟩ `,` ⋯ `,` ⟨*lab*⟩ `))?` ⟨*sqlselect*⟩
⟨*insertRow*⟩	::=	`(` ⟨*insertVal*⟩ `,` ⋯ `,` ⟨*insertVal*⟩ `)`
⟨*insertVal*⟩	::=	⟨*sqlexp*⟩ | `default`
⟨*insertVar*⟩	::=	⟨*vid*⟩ | `op` ⟨*longvid*⟩

The syntax is restricted by the following rules:

- The labels of `(` ⟨*lab*⟩ `,` ⋯ `,` ⟨*lab*⟩ `)` must be distinct.
- The number of ⟨*insertVal*⟩ in ⟨*insertRow*⟩ must be equal to the number of labels in `(` ⟨*lab*⟩ `,` ⋯ `,` ⟨*lab*⟩ `)`.

An INSERT command inserts the rows in either the `values` subclause or the result of the ⟨*sqlselect*⟩ query into the designated table. If an SML# variable ⟨*vid*⟩ or `op` ⟨*longvid*⟩ is written in the `values` subclause, the list of the records consisting of labels ⟨lab_1⟩, ⋯, ⟨lab_n⟩ denoted by the variable is inserted into the table. If the rows to be inserted does not cover all the columns in the designated table, the columns not covered are filled with the default values, which is specified when the table is created. If a column is specified as `default` in a row of the `values` clause, the default value is also inserted into that column. If a default value is needed but no default value is specified, the database server causes an runtime error and the `SQL.Exec` exception is raised in SML#.

The typing rules are the following:

- ⟨*vid*⟩ must be of the `SQL.db` type that has at least the table name ⟨*lab*⟩.

- If there is `(` ⟨lab_1⟩ `,` $\cdots$ `,` ⟨lab_n⟩ `)`, either the table `#` ⟨*vid*⟩ `.` ⟨*lab*⟩ refers to or the ⟨*sqlselect*⟩ query must have at least columns ⟨lab_1⟩, ..., ⟨lab_n⟩. Other columns may be included in it. for ⟨*sqlselect*⟩, the column sets of its result and the designated table may be different.

- If no `(` ⟨lab_1⟩ `,` $\cdots$ `,` ⟨lab_n⟩ `)` appears, the column set of the result of the ⟨*sqlselect*⟩ query and the designated table must be identical.

- The type of the i-th expression ⟨$sqlexp_i$⟩ in the `values` columns must be $(\{\}, w) \triangleright \tau_i$ where τ_i is the type of the ⟨lab_i⟩ column of the designated table.

22.6.2 UPDATE Commands

The syntax of the UPDATE command is the following:

⟨*sqlupdate*⟩ ::= `update #` ⟨*vid*⟩ `.` ⟨*lab*⟩
`set` ⟨*updateRow*⟩ `,` $\cdots$ `,` ⟨*updateRow*⟩
`(` ⟨*sqlWhereClause*⟩ `)?`
⟨*updateRow*⟩ ::= ⟨*lab*⟩ `=` ⟨*sqlexp*⟩

The syntax is restricted by the following rules:

- All the ⟨*lab*⟩s of ⟨*updateRows*⟩ must be distinct.

An UPDATE command updates the rows in the designated table and matched with the condition of the WHERE clause with the values of the SET clause. The columns not specified in the SET clause are left original. The expressions in the SET clauses may include column references of the table ⟨*lab*⟩ in order to refer to the original value.

The typing rules are the following:

- ⟨*vid*⟩ must be of the `SQL.db` type that has at least the table name ⟨*lab*⟩.

- The column ⟨lab_i⟩ of each ⟨$updateRow_i$⟩ must be a column in the designated table. The table may have other columns.

- The type of ⟨$sqlexp_i$⟩ of each ⟨$updateRow_i$⟩ must be $(\{$`⟨`lab`⟩ : `$\tau\}, w) \triangleright \tau_i$ where τ is the type of the designated table and τ_i is the type of the ⟨lab_i⟩ column in τ.

- If there exists ⟨*sqlWhereClause*⟩, it must be of type `(`τ `->` τ`, w) SQL.whr` where τ is the type of the designated table.

22.6.3 DELETE commands

The following is the syntax of the DELETE commands:

⟨*sqldelete*⟩ ::= `delete from #` ⟨*vid*⟩ `.` ⟨*lab*⟩ `(` ⟨*sqlWhereClause*⟩ `)?`

An DELETE commands deletes the rows matched with the condition of the WHERE clause from the designated table.

The typing rules are the following:

- ⟨*vid*⟩ must be of the `SQL.db` type that has at least the table name ⟨*lab*⟩.

- If there exists ⟨*sqlWhereClause*⟩, its type must be of `(`τ `->` τ`, w) SQL.whr` where τ is the type of the designated table.

22.6.4 BEGIN, COMMIT, and ROLLBACK commands

The following transaction control commands are available in SML#:

⟨*sqltransaction*⟩ ::= `begin` start a transaction
| `commit` commit a transaction
| `rollback` abort a transaction

22.7 SQL execution function expressions

An SQL execution function `_sql` ⟨*pat*⟩ `=>` ⟨*sqlfn*⟩ generates a function that executes the SQL command ⟨*sqlfn*⟩ on a database server. By applying this function to a connection handle to a database, the ⟨*sqlfn*⟩ expression is evaluated and the resulting SQL command is sent to the server. This function returns the execution result if the execusion is successfully done, or raises the `SQL.Exec` exception otherwise. This function behaves as the following:

1. It receives a connection handle of the τ `SQL.conn` type as its argument.

2. It obtains an database instance of the `(`τ`,` w`) SQL.db` type from the connection handle and binds the pattern ⟨*pat*⟩ to it.

3. It evaluates the expression ⟨*sqlfn*⟩ and obtains the command of type `(`τ'`,` w`) SQL.command`. Exceptionally, if ⟨*sql*⟩ is of the form ⟨*sqlselect*⟩, it interprets an SELECT query of type `(`τ`,` w`) SQL.query` as an SQL command of type `(`τ `SQL.cursor,` w`) SQL.command`.

4. It serializes the command and sends it to the server.

5. If succeeded, it returns the result of type τ'.

As seen in these steps, the type of the execution function is τ `SQL.conn ->` τ' when the type of the command is `(`τ`,` w`) SQL.db ->` `(`τ'`,` w`) SQL.command`. Typically, τ' is either `SQL.cursor` if ⟨*sqlfn*⟩ is ⟨*sqlselect*⟩, or `unit` otherwise.

The most simple way to execute an SQL query or command is to write it in an SQL execution function directly. For example,

```
val q = _sql db => select #t.name as name, #t.age as age
                   from #db.employee as t
                   where #t.salary >= 300
                   order by #.age
```

is the function `q` that execute the query. To execute the query whose clauses are independently constructed, use the `select...(`⟨*exp*⟩`)` notation in the body of an SQL execution function, as seen in the following:

```
val s = _sql select #t.name as name, #t.age as age
val f = fn db => _sql from #db.employee as t
val g = fn db => select...(s) from...(f db)
val q = _sql db => select...(q db)
```

For SQL commands, write it directly in an SQL execution function. For example,

```
val w = fn () => _sql where #employee.name = "Taro"
val c = fn db => _sql update #db.employee
                      set āge = #employee.age + 1,
                      salary = #employee.salary + 100
                      where...(w ())
val q = _sql db => ...(c db)
```

The typing rules of the SQL execution functions are the following:

- In the type of ⟨*sqlfn*⟩, which is either `(`τ`,`w`) SQL.command` or `(`τ'`,`w`) SQL.query`, the w must be the type variable occuring only in the type of ⟨*sqlfn*⟩.

This restriction is introduced to avoid queries across multiple databases. For example, the following causes a type error:

```
# fun f exp = _sql db => select #e.name, (...exp) from #db.employee as e;
(interactive):1.12-1.65 Error:
  (type inference 067) User type variable cannot be generalized: '$h
```

The formal reason of this error is that the w of `(`τ`,`w`) SQL.query`, which is the type of `select` $\cdots$, is also used in the type of the `exp` argument, `(`τ'`,`w`) SQL.exp`, since `exp` of the argument of `f` is included in `select`, and therefore the above restriction is violated. From the practical perspective, this is an error because the `exp` argument may be an SQL value expression related to the database different from the `db` of `_sql db =>` $\cdots$. To understand the situation, consider another following function that calls `f`:

```
fun badExample conn1 conn2 =
    (_sql db2 =>
          select...((f _sql((select #t1.c1 from #db2.t1)) conn1;
                     _sql(select #t2.c2 from #db2.t2))))
      conn2
```

If `f` is a polymorphic function, the `badExample` function is also typed regardless of the above type restriction. The `badExample` function receives two connection handle `conn1` and `conn2`, which point to possibly different databases, and executes the query of the function `f` and another query on `conn1` and `conn2`, respectively. Its strange behavior is that it executes the query of `f` in an SQL execution function. In addition, it applies `f` to a subquery on `db2` of `conn2`, not `conn1`. Concequently, the query executed in `f` refers to the two databases, one of which is referred to as `db` in `f`, and another of which is referred to as `db2` in `badExample`. Such a query cannot be executed.

In practice, the place in which an SQL execution function can place is limited due to this restriction. In particular, as seen in the above `f`, a function that executes a given query as an argument is not allowed. To avoid this limitation, you should distinguish between query construction functions and query execution functions. For example, if the above `f` is rewritten so that it constructs a query rather than executes it as follow, then the type error is avoided:

```
fun f exp = fn db => _sql select #e.name, (...exp) from #db.employee as e
val q = _sql db => select...(f _sql(#e.saraly) db)
```

The `SQL.Exec` exception is raised in the following situations:

1. Constraint violation other than NOT NULL.
2. Overflow of integers or strings.
3. Division by zero.
4. The use of an SQL syntax that SML# supports but the database server cannot interpret.

Notes on 4.: the SQL syntax of SML# is designed based on the SQL99 standard extended with popular extensions among major database engines and natural extension in the sence of functional programming langauges. The compliance with the standard SQL and the detail of interpretation rules of SQL queries may depend on the implementation of the database servers. Therefore, the programmer should not use SML#'s SQL syntax with no limitation but choose its subset in accordance with the manual of the database server to be used. The following are such incompatibilities known at the time when this manual is authored:

- PostgreSQL ignores `AS` in a FROM clause if the `AS` occur in a join expression labeled with `AS`. For example, the following query causes an error since `x` is undefined:

  ```
  SELECT x.col FROM (a AS x NATURAL JOIN b AS y) AS z
  ```

- SQLite3 does not support the `group by ()` notation, which is introduced in SQL99. Use `group by ""` instead.

22.8 SQL Library: The SQL Structure

SML# provides the types and functions related to SQL as a library. All of them are included in the `SQL` structure provided by the `sql.smi` interface file. An SML# source file that uses the SQL feature must include the following lines in its interface file so that it refers to `sql.smi`:

```
_require "sql.smi"
```

The following is the signature of the `SQL` structure:

```
structure SQL : sig
  type bool3
  type numeric
  type decimal = numeric
  type backend
  type 'a server
  type 'a conn
  type 'a cursor
  type ('a, 'b) exp
  type ('a, 'b) whr
  type ('a, 'b) from
  type ('a, 'b) orderby
  type ('a, 'b, 'c) select
  type ('a, 'b) query
  type ('a, 'b) command
  type ('a, 'b) db
  exception Exec
  exception Connect
  exception Link
  val postgresql : string -> backend
  val mysql : string -> backend
  val odbc : string -> backend
  val sqlite3 : string -> backend
  structure SQLite3 : (see Section 22.8.1)
  val connect : 'a server -> 'a conn
  val connectAndCreate : 'a server -> 'a conn
  val closeConn : 'a conn -> unit
  val fetch : 'a cursor -> 'a option
  val fetchAll : 'a cursor -> 'a list
  val closeCursor : 'a cursor -> unit
  val queryCommand : ('a list, 'b) query -> ('a cursor, 'b) command
  val toy : (('a, 'c) db -> ('b, 'c) query) -> 'a -> 'b
  val commandToString : (('a,'c) db -> ('b,'c) command) -> string
  val queryToString : (('a,'c) db -> ('b,'c) query) -> string
  val expToString : ('a,'c) exp -> string
  Structure Op : (see Section 22.9)
  Structure Numeric : see Section 22.9
  Structure Decimal = Numeric
end
```

These definitions are categorized by their purposes. Each of the following subsections describes the definitions belonging to a category.

22.8.1 Connecting to a database server

- `exception Connect of string`

  The exception of a database connection error. `string` is the error message.

- `exception Link of string`

  The exception indicating the mismatch of database schema and SML# programs. `string` is the error message.

- `type backend`

  The type of the untyped connection information to a database, which is a part of the `_sqlserver` expression. One of the following function can be used to write an expression of this type.

- `val postgresql : string -> backend`

`SQL.postgresql` *param* returns an connection information to a PostgreSQL server. The string *param* is the connection string of libpq, the PostgreSQL library. See the PostgreSQL manual for defails of the connection string. If *param* is not valid, the `SQL.Connect` exception is raised.

The correspondence between the PostgreSQL and SML# types is the following:

PostgreSQL	SML#
`INT`, `INT4`	`int`
`BOOLEAN`	`bool`
`TEXT`, `VARCHAR`	`string`
`FLOAT8`	`real`
`FLOAT4`	`real32`

- `val mysql : string -> backend`

  `SQL.mysql` *param* returns a connection information to a MySQL server. The string *param* consists of the sequences of "key=value" separated by whitespaces. The keys avaliable and their meanings are the following:

Key	Description
`host`	The hostname of a MySQL server
`port`	The port number of a MySQL server
`user`	The user name to log in a MySQL server
`password`	The password of the user
`dbname`	The name of the target database
`unix_socket`	The filename of a UNIX socket
`flags`	The flags of the communication protocol in decimal

  `dbname` is mandatory. See the MySQL manual for details of these parameters. If *param* is not valid, the `SQL.Connect` exception is raised.

  The correspondence between the MySQL and SML# types are the following:

MySQL	SML#
`TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT`	`int`
`TINYTEXT`, `TEXT`, `VARCHAR`	`string`
`DOUBLE`	`real`
`FLOAT`	`real32`

- `val sqlite3 : string -> backend`
  `val sqlite3' : SQL.SQLite3.flags * string -> backend`

  `SQL.sqlite3` *filename* and `SQL.sqlite3 (`*flags*`,` *filename*`)` return a connection information to an SQLite3 database file. The string *filename* indicates the filename. Note that SQLite3 interprets the filenames beginning with ":" in a special way.

  flags is a record consisting of the following four fields:

  - `mode`: the open mode of the file, which is one of the following:
    - `SQL.SQLite3.SQLITE_OPEN_READONLY`
    - `SQL.SQLite3.SQLITE_OPEN_READWRITE`
    - `SQL.SQLite3.SQLITE_OPEN_READWRITE_CREATE`
  - `threading`: the threading mode, which is one of the following:
    - `SQL.SQLite3.SQLITE_OPEN_NOMUTEX`
    - `SQL.SQLite3.SQLITE_OPEN_FULLMUTEX`
  - `cache`: the cache mode, which is one of the following:
    - `SQL.SQLite3.SQLITE_OPEN_SHAREDCACHE`
    - `SQL.SQLite3.SQLITE_OPEN_PRIVATECACHE`
  - `uri`: the way to interpret the filename, which may be the following:
    - `SQL.SQLite3.SQLITE_OPEN_URI`

See the SQLite3 C/C++ API manual for details of these flags. These constants are defined in the `SQL.SQLite3` structure. In addition, `SQL.SQLite3` provides the default flag `SQL.SQLite3.flags`, which is used if *flags* is omitted. A partially modified version of `SQL.SQLite3.flags` can be obatained by using the field update expression.

The correspondence between SQLite3's type affinities and SML#'s type is the following:

Type affinity	SML#
`INT`	`int`
`REAL`	`real`
`STRING`	`string`
`NUMERIC`	`numeric`
`BLOB`	(unsupported)

Each type of columns specified in CREATE TABLE statement is interpreted to a type affinity as described in the SQLite3 manual.

- `val odbc : string -> backen`

  `SQL.odbc` *param* returns a connection information to an ODBC server. The string *param* consists of a DSN name, user name, and password in this order separated by whitespaces. If *param* is not valid, the `SQL.Connect` exception is raised.

  The correspondence between the ODBC and SML# types is the following

ODBC	SML#
`CHAR`	`string`
`INTEGER`, `SMALLINT`	`int`
`FLOAT`	`real32`
`DOUBLE`	`real`
`VARCHAR`, `LONGVARCHAR`, `NVARCHAR`	`string`

- `val connect : 'a server -> 'a conn`

  `SQL.connect` *server* establishes a connection to the server indicated by the connection description *server*. If the connection is established and the schema of the connected database subsumes the schema represented by *server*, it returns a connection handle. If a connection error occurs, the `SQL.Exec` exception is raised. If the two schemas are not matched, the `SQL.Link` exception is raised.

  The type of the connection description *server* represts the type of the tables and views that the SML# program deals with through this connection. `SQL.connect` checks the system catalog of the database so that all tables and views in the type of *server* exists in the database. The names of tables and views are case-insensitive during this check. The database may contain tables and views other than those specified in the type of *server*. In contrast, for each table in *server*, its column set must exactly matches with the actual table definition.

  At the first time to connect to a database server, an external library is dynamically linked according to the kind of the database server. The default name of such libraries are hard-coded. If the library name is not appropriate, it can be changed by setting environment variables. The following table shows the default name and environment vairable name for each server kind:

Database	Library name	Environment variable
PostgreSQL	libpq.so.5	`SMLSHARP_LIBPQ`
MySQL	libmysqlclient.16.so	`SMLSHARP_LIBMYSQLCLIENT`
ODBC	libodbc.so.2	`SMLSHARP_LIBODBC`
SQLite3	libsqlite3.so.0	`SMLSHARP_LIBSQLITE3`

- `val connectAndCreate : 'a server -> 'a conn`

  Same as `SQL.connect` except for the following: `SQL.connectAndCreate` creates the tables that are indicated in the argument type but do not exist in the database by issueing the `CREATE TABLE` commands. If a table to be created includes an unsupported type, it raises the `SQL.Link` exception. If a `CREATE TABLE` command fails, it raises the `SQL.Exec` exception.

- `val closeConn : 'a conn -> unit`

  `SQL.closeConn` *conn* closes a database connection. Any connection established by `SQL.connect` must be closed by `SQL.closeConn`.

22.8.2 executing SQL queries and retrieving their results

- `exception Exec of string`

  The exception indicating that an error occur during a query execution on a database server. `string` is the error message.

- `val fetch : 'a cursor -> 'a option`

  `SQL.fetch` *cursor* reads one row pointed by the cursor *cursor* and move the cursor to the next row. If the cursor reaches to the end of a table, it returns `NONE`. If the cursor is already closed, the `SQL.Exec` exception is raised.

- `val fetchAll : 'a cursor -> 'a list`

  `SQL.fetchAll` *cursor* reads all rows between the cursor *cursor* and the end of the table and closes the cursor. If the cursor is already closed, the `SQL.Exec` exception is raised.

- `closeCursor : 'a cursor -> unit`

  `closeCursor` *cursor* closes the given cursor. All cursors must be closed by this function or `fetchAll`.

22.8.3 Utilities for SQL Queries

- `val queryCommand : ('a list, 'b) query -> ('a cursor, 'b) command`

  `SQL.queryCommand` *query* converts the given SELECT query to a SQL command. This function performs the same thing as what an SQL execution function `_sql` ⟨*pat*⟩ `=> select...(`⟨*exp*⟩`)` does for a query (see Section 22.7) except for the query execution.

- `val toy : (('a, 'c) db -> ('b, 'c) query) -> 'a -> 'b`

  `SQL.toy` *query data* regards *data* as a database and evaluates the query *query* on it. The evaluation is carried out in SML# without involving any server communication. This function executes a toy program that the SML# compiler generates for the typecheck of SQL queries. Note that the performance of the toy program is not considered and therefore this function may be seriously slow.

- `val commandToString : (('a,'c) db -> ('b,'c) command) -> string`

  `SQL.commandToString` *command* returns the serialized string of the SQL command the function *command* returns. The string this function returns is identical to the string sent to the server when the command is executed.

- `val queryToString : (('a,'c) db -> ('b,'c) query) -> string`

  `SQL.queryToString` *query* returns the serialized string of the SELECT query the function *query* returns. The string this function returns is identical to the string sent to the server when the query is executed.

- `val expToString : ('a,'c) exp -> string`

  `SQL.expToString` *exp* returns the serialized string of the SQL value expression *exp*.

22.9 SQL Library: The SQL.Op structure

The `SQL.Op` structure provides SQL's infix operators, aggregate functions, and other utilities, all of which are used for constructing SQL value expressions. In SML#'s SQL value expressions, the infixity of some identifiers are declared as follows:

```
infix 7 %
infix 5 like ||
nonfix mod
```

In an SQL value expression that is not evaluated by SML#, SQL functions and operators defined in this structure can be used without any structure prefix.

Almost all of functions defined in `SQL.Op` is overloaded on multiple SQL basic types (see Section 22.1). In the description, the overloaded type variables and their ranges are indicated by their names as follows:

- `'sql` is one of the SQL basic types or their `option` types.
- `'sqlopt` is the `option` type of an SQL basic type.
- `'num` is one of the SQL basic numeric types or their `option` types.
- `'str` is either `string` or `string option`.
- Other type variables range over the set of all types.

The signature of the `SQL.Op` structure is the following:

```
structure SQL : sig
  ...
  structure Op : sig
    val Some : 'a -> 'a option
    val Part : 'a option list -> 'a list
    val Num : 'num -> numeric option
    val + : 'num * 'num -> 'num
    val - : 'num * 'num -> 'num
    val * : 'num * 'num -> 'num
    val / : 'num * 'num -> 'num
    val mod : 'num * 'num -> 'num
    val ~: 'num -> 'num
    val abs : 'num -> 'num
    val < : 'sql * 'sql -> bool3
    val > : 'sql * 'sql -> bool3
    val <= : 'sql * 'sql -> bool3
    val >= : 'sql * 'sql -> bool3
    val = : 'sql * 'sql -> bool3
    val <> : 'sql * 'sql -> bool3
    val || : 'str * 'str -> 'str
    val like : 'str * 'str -> bool3
    val nullif : 'sqlopt * 'sqlopt -> 'sqlopt
    val coalesce : 'b option * 'b -> 'b
    val coalesce' : 'b option * 'b option -> 'b option
    val count : 'sql list -> int
    val avg : 'num list -> numeric option
    val sum : 'num list -> 'num option
    val sum' : 'num option list -> 'num option
    val min : 'sql list -> 'sql option
    val min' : 'sql option list -> 'sql option
    val max : 'sql list -> 'sql option
    val max' : 'sql option list -> 'sql option
  end
end
```

Each of the following subsections describes the definitions for each category of their purposes.

22.9.1 Workarounds for type inconsistencies

SML# provides the following functions corresponding to implicit type cast. See Subsection 22.4.5 for details.

- `val Some : 'a -> 'a option`

- `val Part : 'a option list -> 'a list`
- `val Num : 'num -> numeric option`

22.9.2 SQL operators and functions

The `SQL.Op` structure provides the following operators. These operators return an SQL value expression that concatinates given expressions with the operator. The comparison is not performed until the constructed query is executed on a database server.

- Comparison operators: `<`, `>`, `<=`, `>=`, `=`, `<>` are provided for any SQL basic types. The type of these operators is

  ```
  'sql * 'sql -> bool3
  ```

- Arithmetic operators: Five infix operators `+`, `-`, `*`, `/`, `%` and two unary operators `~`, `abs` are provided for any SQL numeric types. In the SQL value expression, `%` is declared as a infix identifier. The type of these operators is either

  ```
  'num * 'num -> 'num
  ```

  or

  ```
  'num -> 'num
  ```

- Modulo operation: Following the standard SQL, the modulo operator `mod` is also provided as a function. Note that some database engines supports only one of `mod` and `%`. In the SQL value expression, `mod` is declared as a nonfix identifier. The type of `mod` is

  ```
  'num * 'num -> 'num
  ```

- String operators: The pattern match operator `like` and string concatination operator `||` are available. Both identifiers are infix operators in SQL value expressions. Their types are the following:

  ```
  val like : 'str * 'str -> bool3
  val || : 'str * 'str -> 'str
  ```

- NULLIF: the `nullif` function of the following type is provided:

  ```
  val nullif : 'sqlopt * 'sqlopt -> 'sqlopt
  ```

  Note that the two arguments must be an `option` type. Use `Some` if needed.

- COALESCE: Two variants `coalesce` and `coalesce'` are provided because of the `option` type.

  ```
  val coalesce : 'b option * 'b -> 'b
  val coalesce' : 'b option * 'b option -> 'b option
  ```

  In the SQL query sent to a server, both functions have the same name COALESCE. The type of `coalesce` is chosen for a paticular use of COALESCE that substitutes NULL value with non-NULL values. Different from the standard SQL, COALESCE with more than two arguments is not supported. Nest `coalesce'` functions for more than two values.

22.9.3 SQL aggregation functions

`count`, `avg`, `sum`, `min`, and `max` are available. Because of the `option` type, `sum`, `min`, and `max` function has two variants such as `sum` and `sum'`. The name sent to a server is same regardless of the variant chosen.

The type of these functions are the following:

```
val count : 'sql list -> int
val avg : 'num list -> numeric option
val sum : 'num list -> 'num option
val sum' : 'num option list -> 'num option
val min : 'sql list -> 'sql option
val min' : 'sql option list -> 'sql option
val max : 'sql list -> 'sql option
val max' : 'sql option list -> 'sql option
```

22.10 SQL Library: The SQL.Numeric Structure

The `SQL.Numeric` structure emulates the `NUMERIC` type, which is the numeric type with the maximum precision in the standard SQL.

The following is its signature:

```
structure SQL : sig
  ...
  structure Numeric : sig
    type num = SQL.numeric
    val toLargeInt : num -> LargeInt.int
    val fromLargeInt : LargeInt.int -> num
    val toLargeReal : num -> LargeReal.real
    val fromLargeReal : LargeReal.real -> num
    val toInt : num -> Int.int
    val fromInt : Int.int -> num
    val toDecimal : num -> IEEEReal.decimal_approx
    val + : num * num -> num
    val - : num * num -> num
    val * : num * num -> num
    val quot : num * num -> num
    val rem : num * num -> num
    val compare : num * num -> order
    val < : num * num -> bool
    val <= : num * num -> bool
    val > : num * num -> bool
    val >= : num * num -> bool
    val ~ : num -> num
    val abs : num -> num
    val toString : num -> string
    val fromString : string -> num option
  end
end
```

The meaning of these functions are same as the functions of the same name defined in `Int` and `IntInf` of the Basic Library. Note that `toString` serializes at most 16,383 digits after the decimal point.

This `SQL.Numeric` structure is provided just for interaction with database systems and not intended to be a general infinite-precision decimal arithmetic library. Its performance is not considered and therefore these functions may be seriously slow.

`SQL.decimal` and `SQL.Decimal` are the aliases of `SQL.numeric` and `SQL.Numeric`, respectively.

22.11 Difference from the standard SQL (Informative)

SML#'s SQL syntax is designed to be similar to the standard SQL as much as possible so that the programmer can use it without learning a new embedded language. However, it is inebitable that some SQL notations cannot be used due to the conflict with the host language syntax. This section summarizes the difference between SML# and SQL. By paying attention to the following, you can exploit the SML#'s SQL feature with the standard notation of SQL.

- Constant literals are written in the SML# notation.
- All the keywords must be in small cases in SML#, whereas the keywords of the standard SQL are case-insensitive,
- The names of tables and columns are case-sensitive in SML#. Only one exception is that names are compared with ignoring the cases when `SQL.connect` checks schema.
- The arguments of function applications does not need to be parenthesized as in SML#, whereas the standard SQL requires the arguments parenthesized.

- The unary minus operator is `~`, not `-`.
- The associativity of infix operators are decided by SML#'s infix declarations.
- The logical conjunction operator `and` may occur only in a parenthesized expression.
- Each column reference must be prefixed with `#`.
- A table name is required in each column reference.
- When an ORDER BY clause refers to a column of the SELECT clause, the reference must be prefixed with `#.`.
- Each table name occuring in a FROM clause or an INSERT, UPDATE, or DELETE command must be prefixed with `#` and an identifier indicating an database instance, which is typically bound by an `_sql` or `fn` expression.

Chapter 23

Declarations of the core language and their interfaces

For each declaration (⟨*decl*⟩), its syntax and the corresponding interface specification are defined, and describes their static and dynamic values.

23.1 val declarations : ⟨*valDecl*⟩

Syntax of val declarations is given below.

⟨*valDecl*⟩	::=	`val` ⟨*tyvarSeq*⟩ ⟨*valBind*⟩
⟨*valBind*⟩	::=	⟨*valBind1*⟩
	|	⟨*valBind1*⟩ `and` ⟨*valBind*⟩
⟨*valBind1*⟩	::=	⟨*pat*⟩ `=` ⟨*exp*⟩

In this declaration, the variables appearing in patterns ⟨*valBind*⟩ must be distinct. These variables are simultaneously defined, whose scope is the entire ⟨*valDecl*⟩ declaration and the declarations that follows.

The type variable declaration ⟨*tyvarSeq*⟩ in ⟨*valDecl*⟩ delimit their scope. These type variables must be generalized at the top-level of each ⟨*valBind1*⟩ declaration.

23.1.1 val declaration interface : ⟨*valSpec*⟩

For each variable defined in a val declaration, the following form of interface specification must be declared.

⟨*valBindInterface*⟩ ::= `val` ⟨*id*⟩ `:` ⟨*ty*⟩

For example, for val declaration

```
val (x,y) = (1,2)
```

the following two interface specification must be given.

```
val x : int
val y : int
```

23.1.2 val declaration evaluation

Val declaration of the form

`val` ⟨pat_1⟩ `=` ⟨exp_1⟩ `and` ⟨pat_2⟩ `=` ⟨exp_2⟩ ⋯

is evaluated in the following two steps.

Structured pattern evaluating

A val declaration with a structured pattern $\langle pat \rangle$ is first transformed to sequence of val declarations for the set $\{x_1, \ldots, x_m\}$ of variables in the pattern $\langle pat \rangle$.

If the pattern $\langle pat \rangle$ does not contain constructor or constant, then the transformation is done by recursively decompose $\langle pat_i \rangle$ and $\langle exp_i \rangle$ pair to a sequence of pairs of a sub-pattern and a sub-expression. This is done in the following steps.

1. Each pattern $\langle pat_i \rangle$ is decomposed into sub-patterns according to the structure of $\langle pat_i \rangle$. When $\langle pat_i \rangle$ has a type constraint, then the corresponding type constraint is attached to each of the decomposed sub-patterns.

2. For each decomposed sub-pattern of $\langle pat_i \rangle$, the corresponding sub-expression is generated from the expression $\langle exp_i \rangle$ by applying the code to extract the corresponding value to $\langle exp_i \rangle$.

3. If a sub-pattern is a layered pattern of the form *id* `as` *pat*, the additional code to bind *id* to the entire value is generated.

4. Finally, the following from of val declarations is generated from the sequence of pairs of a variable and an expression obtained from the above transformation steps.

   `val` x_1 $(: \tau_1)?$ `=` exp_1
   ...
   `and` x_m $(: \tau_m)?$ `=` exp_m

This decomposition process enables the variables in structured patterns to have rank-1 polymorphic types as far as possible. However, the above transformation cannot be applied to val declarations with constructors and constants, since these val declarations may raise exception at runtime. A val declaration containing constructors and constants is transformed to the following val declaration with a variable.

```
val X = case ( ⟨exp1⟩ , ..., ⟨expn⟩ ) of
          ( ⟨pat1⟩ , ..., ⟨patn⟩ ) => (x1,...,xm)
        | _ => raise Bind
val x1 = #1 X
  ...
and xm = #m X
```

23.1.3 Example of val declarations and interface

The following are simple examples in the interactive mode.

```
# val (x,y) = (print "SML#\n", fn x => fn y => (x,y));
SML#
val x = () : unit
val y = fn : ['a. 'a -> ['b. 'b -> 'a * 'b]]

# val (z, w, 1) = (print "SML#\n", fn x => fn y => (x,y), 1);
(interactive):2.8-2.8 Warning:
  (type inference 065) dummy type variable(s) are introduced due to value
  restriction in: y
(interactive):2.4-2.57 Warning: binding not exhaustive
  (x, y, 1) => ...
SML#
val z = () : unit
val w = fn : fn : ?X7 -> ?X6 -> ?X7 * ?X6
```

The following is a simple example of a source file and an interface file in separate compilation.

Version.sml file:

```
val (version, releaseDate) = ("4.2.0", "2025-03-24")
```

Version.smi file:

```
val version : string
val releaseDate : string
```

23.2 Function declarations : ⟨*valRecDecl*⟩ , ⟨*funDecl*⟩

Syntax of function declarations has the following two forms ⟨*valRecDecl*⟩ and ⟨*funDecl*⟩ .

⟨*valRecDecl*⟩ ::= `val rec` ⟨*tyvarSeq*⟩ ⟨*valBind*⟩
⟨*funDecl*⟩ ::= `fun` ⟨*tyvarSeq*⟩ ⟨*funBind*⟩
⟨*funBind*⟩ ::= ⟨*funBind1*⟩
| ⟨*funBind1*⟩ `and` ⟨*funBind*⟩
⟨*funBind1*⟩ ::= (`op`)? ⟨*vid*⟩ ⟨*atpat*$_{11}$⟩ ⋯ ⟨*atpat*$_{1n}$⟩ (`:` ⟨*ty*⟩)? `=` ⟨*exp*$_1$⟩ $(m, n \geq 1)$
| (`op`)? ⟨*vid*⟩ ⟨*atpat*$_{21}$⟩ ⋯ ⟨*atpat*$_{2n}$⟩ (`:` ⟨*ty*⟩)? `=` ⟨*exp*$_2$⟩
| ⋯
| (`op`)? ⟨*vid*⟩ ⟨*atpat*$_{m1}$⟩ ⋯ ⟨*atpat*$_{mn}$⟩ (`:` ⟨*ty*⟩)? `=` ⟨*exp*$_m$⟩

These declarations define mutual recursive functions. The scope of the function names being defined is the entire declaration and the declarations that follow.

val declarations of ⟨*valBind*⟩ in ⟨*valRecDecl*⟩ are restricted to those that have the following syntactic form:

⟨*vid*⟩ `=` `fn` expression

The identifiers (function names) ⟨*vid*⟩ appearing in the same ⟨*funBind1*⟩ must be the same, and function names appearing in different ⟨*funBind1*⟩s must be pair-wise distinct. As in ⟨*valBind1*⟩ , variables contained in patterns ⟨*pat*⟩ in the same ⟨*funBind1*⟩ must be pair-wise distinct.

In evaluation of these declarations, function names (⟨*vid*⟩) are bound to the same (static and dynamic) values as the corresponding functions.

The type variable declaration ⟨*tyvarSeq*⟩ in ⟨*valBind*⟩ and ⟨*funDecl*⟩ delimit their scope. These type variables must be generalized at the top-level of each ⟨*valBind1*⟩ of ⟨*valBind*⟩ and ⟨*funBind1*⟩ .

23.2.1 Function declaration interface

Function declaration interface is the same as val declaration interface. Each function name defined in the function declaration and its type are specified in the same syntax of val declaration specification. The following is an example of a source file and an interface file containing function declarations.

`Bool.sml` file:

```
fun not true = false
  | not false = true
fun toString true = "true"
  | toString false = "false"
```

`Bool.smi` file:

```
val not : bool -> bool
val toString : bool -> string
```

23.3 datatype declaration : ⟨*datatypeDecl*⟩

datatype declarations define new type constructors. Its syntax is given below.

⟨*datatypeDecl*⟩ ::= `datatype` ⟨*datbind*⟩ (`withtype` ⟨*tybind*⟩)?
⟨*datbind*⟩ ::= (⟨*tyvarSeq*⟩)? ⟨*tycon*⟩ `=` ⟨*conbind*⟩ (`and` ⟨*datbind*⟩)?
⟨*conbind*⟩ ::= (`op`)? ⟨*vid*⟩ (`of` ⟨*ty*⟩)? (`|` ⟨*conbind*⟩)?
⟨*tybind*⟩ ::= ⟨*tyvarSeq*⟩ ⟨*tycon*⟩ `=` ⟨*ty*⟩ (`and` ⟨*tybind*⟩)?

The set of type constructor names ⟨*tycon*⟩ must be pair-wise distinct, and the data constructor names in the same ⟨*conbind*⟩ must be pair-wise distinct. This defines the set of mutually recursive new type constructors ⟨*tycon*⟩, and data constructors ⟨*vid*⟩. If ⟨*tycon*⟩ has optional (⟨*tyvarSeq*⟩)? declaration, then it is a polymorphic type constructor with type parameters ⟨*tyvarSeq*⟩. The scope of ⟨*tycon*⟩ include the entire ⟨*datatypeDecl*⟩.

23.3.1 datatype declaration interface

Interface of datatype declaration is either datatype specification or opaque type specification. Syntax of datatype specification is the same as datatype declarations. Opaque type specification has the following syntax.

⟨*opequeTypeSpec*⟩ ::= `type` (⟨*tyvarSeq*⟩)? ⟨*tycon*⟩ `(` `=` ⟨*runtimeTypeSpec*⟩ `)`
⟨*runtimeTypeSpec*⟩ := `unit` | `contag` | `boxed`

A datatype specification defines the type constructors and data constructors of the corresponding datatype declaration, and makes them available to the compilation units that reference this interface through `_require`. An opaque type specification must defines the type constructor of the corresponding datatype declaration as an opaque type, and makes it available to the compilation units that reference this interface through `_require`. The operand of a opaque type specification specifies the runtime representation of values of the type. It is either `unit`, `contag`, or `boxed`. `unit` and `contag` indicates that all the datatype constructors have no argument. `unit` indicates that the type consists only of a single constructor. `contag` means that there are multiple constructors. `boxed` indicates that the runtime representation is a pointer. This must be specified for a datatype constructor that has a constructor with augment.

23.3.2 Examples

The following are examples of datatype declarations and their interface.

`Data.sml` file:

```
datatype 'a list = nil | :: of 'a * 'a list
datatype 'a queue = QUEUE of 'a list * 'a list
```

`Data.smi` file:

```
datatype 'a list = nil | :: of 'a * 'a list
type 'a queue (= boxed)
```

23.4 Type declaration : ⟨*typDecl*⟩

Type declaration define a name of a type. Its syntax is given below.

⟨*typeDecl*⟩ := `type` ⟨*typbind*⟩
⟨*tybind*⟩ ::= ⟨*tyvarSeq*⟩ ⟨*tycon*⟩ `=` ⟨*ty*⟩ (`and` ⟨*tybind*⟩)?

23.4.1 type specification : ⟨*typSpec*⟩

Interface of type declaration is either type specification or opaque type specification. Syntax of type specification is the same as type declaration. Opaque type specification has the following syntax.

⟨*opequeTypeSpec*⟩ := `type` (⟨*tyvarSeq*⟩)? ⟨*tycon*⟩ `(` `=` ⟨*runtimeTypeSpec*⟩ `)`
⟨*runtimeTypeSpec*⟩ := ⟨*tycon*⟩ | `{}` | `*` | `->`

If the type is implemented by a record type, tuple type, or function type, ⟨*runtimeTypeSpec*⟩ must be `{}`, `*`, or `->`, respectively. Otherwise, ⟨*runtimeTypeSpec*⟩ must be the type constructor that implements the type. ⟨*runtimeTypeSpec*⟩ may not be identical to the implmentation type if the runtime representation of ⟨*runtimeTypeSpec*⟩ is matched with that of the implementation type. See Chapter 29 for details of the runtime representation.

23.4.2 Examples

The following are examples of type declarations and their interface.

`Data.sml` file:

```
type 'a set = 'a list
type 'a queue = 'a list * 'a list
type index = int
type id = int
```

`Data.smi` file:

```
type 'a set (= list)
type 'a queue (= *)
type index = int
type id (= int)
```

23.5 Exception declaration : ⟨*exnDecl*⟩

Syntax of exception declaration is given below.

⟨*exnDecl*⟩ ::= `exception` ⟨*exbind*⟩
⟨*exbind*⟩ ::= (`op`)? ⟨*vid*⟩ (`of` ⟨*ty*⟩)? (`and` ⟨*exbind*⟩)?

This defines an exception constructor optionally with an argument of type ⟨*ty*⟩.

23.5.1 Exception specification : ⟨*exnSpec*⟩

Syntax of exception specification is the same as exception declaration.

23.5.2 Examples

The following are examples of exception declarations and their interface.

`Data.sml` file:

```
exception Fail of string
```

`Data.smi` file:

```
exception Fail of string
```

Chapter 24

Module language declarations and interface

For each declaration (⟨*decl*⟩), its syntax and the corresponding interface specification are defined, and describes their static and dynamic values.

24.1 Structure declarations : ⟨*strDecl*⟩

Syntax for structure declaration is given below.

⟨*strDecl*⟩	:=	`structure` ⟨*strbind*⟩
⟨*strbind*⟩	::=	⟨*strid*⟩ `=` ⟨*strexp*⟩ (`and` ⟨*strbind*⟩)?
	|	⟨*strid*⟩ `:` ⟨*sigexp*⟩ `=` ⟨*strexp*⟩ (`and` ⟨*strbind*⟩)?
	|	⟨*strid*⟩ `:>` ⟨*sigexp*⟩ `=` ⟨*strexp*⟩ (`and` ⟨*strbind*⟩)?

If there are multiple bindings separated with `and` then the structure names are bound simultaneously, whose scope is the rest of declarations that follow this declaration. So, in the following example, `y` is bound to `2` and not `1`.

```
structure A = struct val a = 1 end;
structure B = struct val b = 1 end;
structure A = struct val a = 2 end and B = struct val b = A.a end;
val x = A.a;
val y = B.b;
```

Binding structure names with signature constraints are translated to binding to structure expressions with signature constraints as follows.

source	translated to
⟨*strid*⟩ `:` ⟨*sigexp*⟩ `=` ⟨*strexp*⟩	⟨*strid*⟩ `=` ⟨*strexp*⟩ `:` ⟨*sigexp*⟩
⟨*strid*⟩ `:>` ⟨*sigexp*⟩ `=` ⟨*strexp*⟩	⟨*strid*⟩ `=` ⟨*strexp*⟩ `:>` ⟨*sigexp*⟩

Evaluation of a structure declaration is done as follows.

1. Evaluate the structure expression and obtain a static type environment Γ and a runtime environment E, as defined in the following section.

2. Obtain Γ' and E' from Γ' and E' by prefixing the long names defined in them with the structure name S, and add them to the current environments.

For example, if structure expression ⟨*strexp*⟩ generates a static type environment $\{longId_1 : \tau_1, ..., longid_n : \tau_n\}$ and a runtime value environment $\{longId_1 : v_1, ..., longid_n : v_n\}$, then evaluating the structure declaration `structure` ⟨*strid*⟩ `=` ⟨*strexp*⟩ has the effect of extending the current type environment and runtime environment with the bindings $\{S.longId_1 : \tau_1, ..., S.longid_n : \tau_n\}$ and $\{S.longId_1 : v_1, ..., S.longid_n : v_n\}$.

24.2 Structure expressions and their evaluation : ⟨*strexp*⟩

Syntax for structure expressions is given below.

⟨*strexp*⟩	::=	`struct` ⟨*strdec*⟩ `end`	
	|	⟨*longStrid*⟩	
	|	⟨*strexp*⟩ `:` ⟨*sigexp*⟩	
	|	⟨*strexp*⟩ `:>` ⟨*sigexp*⟩	
	|	⟨*funid*⟩ `(` ⟨*strid*⟩ `:` ⟨*sigexp*⟩ `)`	
	|	⟨*funid*⟩ `(` ⟨*strdec*⟩ `)`	
⟨*strdec*⟩	::=	⟨*decl*⟩	core language declaration
	|	`structure` ⟨*strbind*⟩	
	|	`local` ⟨*strdec*⟩ `in` ⟨*strdec*⟩ `end`	
	|	⟨*strdec*⟩ `(;)?` ⟨*strdec*⟩	

Each of the structure expressions are evaluated as follows.

- Basic structure expression : `struct` ⟨*strdec*⟩ `end`.

  Evaluation of a basic structure expression is done by evaluating the sequence of declarations ⟨*strdec*⟩. For core language declarations, bindings of variables and type constructors are generated according to the definition in Chapter 23. For structure declarations, long name bindings are generated according to the evaluation rule stated above.

- Long structure identifier : ⟨*longStrId*⟩.

  Evaluation of this yields the type environment and the runtime environment ⟨*longStrId*⟩ is bound to in the current environments.

- Structure expressions with signature constraints. (⟨*strexp*⟩ `:` ⟨*sigexp*⟩, ⟨*strexp*⟩ `:>` ⟨*sigexp*⟩)

  Evaluation of a structure expression with signature constraint is done as follows.

  1. Evaluate the structure expression and obtain a static type environment Γ.
  2. Evaluate the signature expression according to the definition in the following subsection (24.3) and obtain a set Σ of static constraints of long names.
  3. For each long name constraint in Σ, check whether the corresponding long name binding exists in Γ and its static value conforms to the constraint.
  4. Obtain a static type environment Γ' by restricting the set of long names to those in Σ.
  5. If the signature constraint is an opaque constraint of the form ⟨*strexp*⟩ `:>` ⟨*sigexp*⟩, then obtain Γ'' from Γ' by replacing each type constructor corresponding to type spec ⟨*typdesc*⟩ in ⟨*sigexp*⟩ with an abstract type constructor.
  6. For constructing a runtime environment, dynamic value generation is the same as the structure expression without signature constraints, but the bound long names are restricted to those specified in Γ''.

24.3 Signature expression : ⟨*sigexp*⟩

The syntax of signature expression is given below.

⟨*sigexp*⟩	::=	`sig` ⟨*spec*⟩ `end`
	|	⟨*sigid*⟩
	|	⟨*sigexp*⟩ `where type` ⟨*tyvarSeq*⟩ ⟨*longTycon*⟩ `=` ⟨*ty*⟩

- Basic signature expression (`sig` ⟨*spec*⟩ `end`)

  It is a list of declaration specification

- Signature name (⟨*sigid*⟩)

  It is a name bound to a signature expression in a top-level signature declaration.

- Signature name with type definitions

  It represent the signature expression obtained from the signature expression bound to ⟨*sigin*⟩ by replacing ⟨*tyvarSeq*⟩ ⟨*longTycon*⟩ with the specified ⟨*ty*⟩ .

The syntax of declaration specification ⟨*spec*⟩ is given below.

⟨*spec*⟩ ::= `val` ⟨*valdesc*⟩
| `type` ⟨*typdesc*⟩
| `eqtype` ⟨*typdesc*⟩
| `datatype` ⟨*datdesc*⟩
| `datatype` ⟨*tycon*⟩ `=` `datatype` ⟨*longTycon*⟩
| `exception` ⟨*exdesc*⟩
| `structure` ⟨*strdesc*⟩
| `include` ⟨*sigexp*⟩
| ⟨*spec*⟩ `sharing type` ⟨$longTycon_1$⟩ `=` $\cdots$ `=` ⟨$longTycon_n$⟩
| ⟨*spec*⟩ (`;`)? ⟨*spec*⟩
|

⟨*valdesc*⟩ ::= ⟨*vid*⟩ `:` ⟨*ty*⟩ (`and` ⟨*valdesc*⟩)?
⟨*typdesc*⟩ ::= ⟨*tyvarSeq*⟩ ⟨*tycon*⟩ (`and` ⟨*typdesc*⟩)?
⟨*datdesc*⟩ ::= ⟨*tyvarSeq*⟩ ⟨*tycon*⟩ `=` ⟨*condesc*⟩ (`and` ⟨*datdesc*⟩)?
⟨*condesc*⟩ ::= ⟨*vid*⟩ (`of` ⟨*ty*⟩)? (`|` ⟨*condesc*⟩)?
⟨*exdesc*⟩ ::= ⟨*vid*⟩ (`of` ⟨*ty*⟩)? (`and` ⟨*exdesc*⟩)?
⟨*strdesc*⟩ ::= ⟨*strid*⟩ `:` ⟨*sigexp*⟩ (`and` ⟨*strdesc*⟩)?

The meaning of each of components is as follows.

- `val` ⟨*valdesc*⟩ . This specifies the type of each variable bound in a val declaration.
- `type` ⟨*typdesc*⟩ . This specifies the existence of a type declaration or datatype declaration in a structure expression.

  When the signature is used as a transparent signature constraint, then this specification makes the type function available. If the matching declaration is a datatype declaration, then this specification makes the type constructor available, but data constructors are not bound.

  When the signature is used as an opaque signature constraint, then this specification changes a type function or a datatype definition to an abstract type constructor.
- `eqtype` ⟨*typdesc*⟩

  This is the same as `type` ⟨*typdesc*⟩ with the additional constraint that the type declared in a structure is restricted to a type that admit equality.
- `datatype` ⟨*datdesc*⟩

  This specifies that a datatype declaration exists, and makes the datatype declaration available.
- `datatype` ⟨*tycon*⟩ `=` `datatype` ⟨*longTycon*⟩

  It specifies that the type constructor bound to the type constructor name ⟨*tycon*⟩ is the same as that bound to the long type constructor name ⟨*longTycon*⟩ .
- `exception` ⟨*exdesc*⟩

  This specifies that an exception declaration exists, and makes the exception declaration available.
- `structure` ⟨*strdesc*⟩

  It specifies that the structure contain a structure declaration of the specified signature.
- `include`

  It expands to the contents (a list of ⟨*spec*⟩) of the signature expression bound to the signature name.
- `sharing type`

  This specifies that the specified set of long type constructor names are bound to the same type constructor.

24.4 Module language interface

As defined in Section 16.2, module language interface consists of t provides for structures (⟨*provideStr*⟩) and for functors (⟨*provideFun*⟩), whose syntax is given below.

⟨*provideStr*⟩	::=	`structure` ⟨*strid*⟩ `=` `struct` ⟨*provideStrdecl*⟩ `end`
⟨*provideStrdecl*⟩	::=	⟨*provideVal*⟩
	|	⟨*provideType*⟩
	|	⟨*provideDatatype*⟩
	|	⟨*provideException*⟩
	|	⟨*provideStr*⟩
⟨*provideFun*⟩	::=	`functor` ⟨*provideFunBind*⟩
⟨*provideFunBind*⟩	::=	⟨*funid*⟩ `(` ⟨*strdesc*⟩ `)` `=` ⟨*provideStrExp*⟩

The following are examples of structure declarations and their interface.

`Bool.sml` file:

```
structure Bool =
struct
  datatype bool = false | true
  fun not true = false
    | not false = true
  fun toString true = "true"
    | toString false = "false"
end
```

`Bool.smi` file:

```
structure Bool =
struct
  datatype bool = false | true
  val not : bool -> bool
  val toString : bool -> string
end
```

Chapter 25

Overview of SML# Libraries

As a functional language, SML# provides various data structures and their operations through a structure declaration containing types and function definitions. Related structures are bundled together in a source file (sml file) with an interface file (smi file) through SML# separate compilation system. In this document, we call a group of structures represented by an interface file as a library. Related libraries are hierarchically organized through `_include` statement of the SML# interface language.

The set of libraries provided by SML# are classified into the following categories.

- Standard ML Basis Library

  This is a standard library for Standard ML language specified in [2]. It provides primitive functions for built-in data types IO primitives, and OS interface. SML# provides all the required structures of Standard ML Basis Library and some optional ones.

- SML# Library

  They include the following support libraries for the SML# advanced features.

  - FFI library
  - SQL library
  - Thread library
  - Reify library

- Miscellaneous utilities

  They provide various useful functions.

  - SML New Jersey Library

- Tool support libraries

  They are libraries for using SML# programming tools in SML# programs. They include the following.

  - smlyacc and smllex support libraries
  - Printer generator SMLFormat support library
  - Unit test tool SMLUnit support Library

In this document, we define a library specification in the following format.

- For the libraries such as those of Standard ML Basis Library for which a formal signature is define, we first show the signature and then for each structure that implements the signature, we give additional type instantiation information that are not represented by the signature.
- For those that have no signature specification, we show their interface information. An interface contains type information for separate compilation that are not represented by the signature.

Chapter 26

Standard ML Basis Library

SML# provide all the required libraries defined in Standard ML Basis Library specifications [2]. All the structures have signature specifications. In the following sections, we first define the formal signatures. For each signature specification, we define the set of structures that implement the signature with additional type information. The details of the semantics of functions, we refer the reader to the specification of the Standard ML Basis Library [2].

The following are the (top-level) signatures and structures provided by SML#.

Signature Name	Structures that implements the signature	section
ARRAY	Array	(26.1)
ARRAY_SLICE	ArraySlice	(26.2)
BIN_IO	BinIO	(26.3)
BOOL	Bool	(26.6)
BYTE	Byte	(26.7)
CHAR	Char	(26.8)
COMMAND_LINE	Commandline	(26.9)
DATE	Date	(26.10)
GENERAL	General	(26.11)
IEEE_REAL	IEEEReal	(26.12)
INTEGER	Int, Int64, Int32, Int16, Int8, Position, LargeInt	(26.14)
INT_INF	IntInf	(26.15)
IO	IO	(26.13)
LIST	List	(26.16)
LIST_PAIR	ListPair	(26.17)
MONO_ARRAY	CharArray, Word8Array	(26.18)
MONO_ARRAY_SLICE	CharArraySlice, Word8ArraySlice	(26.19)
MONO_VECTOR	CharVector, Word8Vector	(26.20)
MONO_VECTOR_SLICE	CharVectorSlice, Word8VectorSlice	(26.21)
OPTION	Option	(26.22)
OS	Os	(26.23)
PRIM_IO	BinPrimIO, TextPrimIO	(26.36)
REAL	Real, Real32, Real64	(26.28)
STRING	String	(26.30)
STRING_CVT	StringCvt	(26.31)
SUBSTRING	Substring	(26.32)
TEXT	Text	(26.33)
TEXT_IO	TextIO	(26.34)
TIME	Time	(26.37)
TIMER	Timer	(26.38)
VECTOR	Vector	(26.39)
VECTOR_SLICE	VectorSlice	(26.40)
WORD	Word, Word64, Word32, Word16, Word8, LargeWord	(26.41)

The set of interface files of all the structures in the basis library described in this chapter are hierarchically organized and are included in the interface file `basis.smi`. Writing the following declaration

in an interface file of a source file makes all the basis library structures and signatures available to the source program.

```
_require "basis.smi"
```

26.1 ARRAY

This structure provides a set of operations for the built-in type τ `array`.

Signature

```
signature ARRAY =
sig
  type 'a array = 'a array
  type 'a vector = 'a Vector.vector
  val all : ('a -> bool) -> 'a array -> bool
  val app : ('a -> unit) -> 'a array -> unit
  val appi : (int * 'a -> unit) -> 'a array -> unit
  val array : int * 'a -> 'a array
  val collate : ('a * 'a -> order) -> 'a array * 'a array -> order
  val copy : {src : 'a array, dst : 'a array, di : int} -> unit
  val copyVec : {src : 'a vector, dst : 'a array, di : int} -> unit
  val exists : ('a -> bool) -> 'a array -> bool
  val find : ('a -> bool) -> 'a array -> 'a option
  val findi : (int * 'a -> bool) -> 'a array -> (int * 'a) option
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
  val foldli : (int * 'a * 'b -> 'b) -> 'b -> 'a array -> 'b
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a array -> 'b
  val foldri : (int * 'a * 'b -> 'b) -> 'b -> 'a array -> 'b
  val fromList : 'a list -> 'a array
  val length : 'a array -> int
  val maxLen : int
  val modify : ('a -> 'a) -> 'a array -> unit
  val modifyi : (int * 'a -> 'a) -> 'a array -> unit
  val sub : 'a array * int -> 'a
  val tabulate : int * (int -> 'a) -> 'a array
  val update : 'a array * int * 'a -> unit
  val vector : 'a array -> 'a vector
end
```

Structures that implement the signature

- `Array:ARRAY`.

26.2 ARRAY_SLICE

It provides a opaque data type for sub-rage of array, array slice, and a set of primitive for the type.

```
signature ARRAY_SLICE =
sig
  type 'a slice
  val length : 'a slice -> int
  val all : ('a -> bool) -> 'a slice -> bool
  val app : ('a -> unit) -> 'a slice -> unit
  val appi : (int * 'a -> unit) -> 'a slice -> unit
  val base : 'a slice -> 'a Array.array * int * int
  val collate : ('a * 'a -> order) -> 'a slice * 'a slice -> order
  val copy : {src : 'a slice, dst : 'a Array.array, di : int} -> unit
```

```
  val copyVec : {src : 'a VectorSlice.slice, dst : 'a Array.array, di : int} -> unit
  val exists : ('a -> bool) -> 'a slice -> bool
  val find : ('a -> bool) -> 'a slice -> 'a option
  val findi : (int * 'a -> bool) -> 'a slice -> (int * 'a) option
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a slice -> 'b
  val foldli : (int * 'a * 'b -> 'b) -> 'b -> 'a slice -> 'b
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a slice -> 'b
  val foldri : (int * 'a * 'b -> 'b) -> 'b -> 'a slice -> 'b
  val full : 'a Array.array -> 'a slice
  val getItem : 'a slice -> ('a * 'a slice) option
  val isEmpty : 'a slice -> bool
  val modify : ('a -> 'a) -> 'a slice -> unit
  val modifyi : (int * 'a -> 'a) -> 'a slice -> unit
  val slice : 'a Array.array * int * int option -> 'a slice
  val sub : 'a slice * int -> 'a
  val subslice : 'a slice * int * int option -> 'a slice
  val update : 'a slice * int * 'a -> unit
  val vector : 'a slice -> 'a Vector.vector
end
```

Structures that implement the signature

- `ArraySlide : ARRAY_SLICE`

  ```
  type 'a slice (= boxed)
  ```

26.3 BIN_IO

This provide binary IO primitives. It is defined as an extension of `IMPERATIVE_IO` (26.4) signature.

```
signature BIN_IO =
sig
  include IMPERATIVE_IO
    where type StreamIO.elem = Word8.word
    where type StreamIO.pos = BinPrimIO.pos
    where type StreamIO.reader = BinPrimIO.reader
    where type StreamIO.writer = BinPrimIO.writer
    where type StreamIO.vector = Word8Vector.vector
  val openAppend : string -> outstream
  val openIn : string -> instream
  val openOut : string -> outstream
end
```

Structures that implement the signature

- `BinIO :> BIN_IO`

  ```
  structure StreamIO = struct
    type elem = word8
    type instream (= boxed)
    type out_pos (= boxed)
    type outstream (= boxed)
    type pos = Position.int
    type reader (= boxed)
    type vector = word8 vector
    type writer (= boxed)
  end
  type elem = word8
  type instream (= boxed)
  ```

```
        type outstream (= boxed)
        type vector = word8 vector
```

26.4 IMPERATIVE_IO

```
signature IMPERATIVE_IO =
sig
  structure StreamIO : STREAM_IO
  type elem = StreamIO.elem
  type instream
  type outstream
  type vector = StreamIO.vector
  val canInput : instream * int -> int option
  val closeIn : instream -> unit
  val closeOut : outstream -> unit
  val endOfStream : instream -> bool
  val flushOut : outstream -> unit
  val getInstream : instream -> StreamIO.instream
  val getOutstream : outstream -> StreamIO.outstream
  val getPosOut : outstream -> StreamIO.out_pos
  val input : instream -> vector
  val input1 : instream -> elem option
  val inputAll : instream -> vector
  val inputN : instream * int -> vector
  val lookahead : instream -> elem option
  val mkInstream : StreamIO.instream -> instream
  val mkOutstream : StreamIO.outstream -> outstream
  val output : outstream * vector -> unit
  val output1 : outstream * elem -> unit
  val setInstream : instream * StreamIO.instream -> unit
  val setOutstream : outstream * StreamIO.outstream -> unit
  val setPosOut : outstream * StreamIO.out_pos -> unit
end
```

Nested signatures

- STREAM_IO(26.5)

26.5 STREAM_IO

```
signature STREAM_IO =
sig
  type elem
  type instream
  type out_pos
  type outstream
  type pos
  type reader
  type vector
  type writer
  val canInput : instream * int -> int option
  val closeIn : instream -> unit
  val closeOut : outstream -> unit
  val endOfStream : instream -> bool
  val filePosIn : instream -> pos
  val filePosOut : out_pos -> pos
```

```
  val flushOut : outstream -> unit
  val getBufferMode : outstream -> IO.buffer_mode
  val getPosOut : outstream -> out_pos
  val getReader : instream -> reader * vector
  val getWriter : outstream -> writer * IO.buffer_mode
  val input : instream -> vector * instream
  val input1 : instream -> (elem * instream) option
  val inputAll : instream -> vector * instream
  val inputN : instream * int -> vector * instream
  val mkInstream : reader * vector -> instream
  val mkOutstream : writer * IO.buffer_mode -> outstream
  val output : outstream * vector -> unit
  val output1 : outstream * elem -> unit
  val setBufferMode : outstream * IO.buffer_mode -> unit
  val setPosOut : out_pos -> outstream
end
```

26.6 BOOL

It provide boolean data type and its primitives.

```
signature BOOL =
sig
  type bool = bool
  val fromString : string -> bool option
  val not : bool -> bool
  val scan : (char, 'a) StringCvt.reader -> (bool, 'a) StringCvt.reader
  val toString : bool -> string
end
```

Structures that implement the signature

- `Bool : BOOL`

26.7 BYTE

A support library for converting character and byte data.

```
signature BYTE =
sig
  val byteToChar : word8 -> char
  val bytesToString : Word8Vector.vector -> string
  val charToByte : char -> word8
  val packString : Word8Array.array * int * substring -> unit
  val stringToBytes : string -> Word8Vector.vector
  val unpackString : Word8ArraySlice.slice -> string
  val unpackStringVec : Word8VectorSlice.slice -> string
end
```

Structures that implement the signature

- `Byte : BYTE`

26.8 CHAR

Provide primitives for character data.

```
signature CHAR =
sig
  eqtype char
  eqtype string
  val < : char * char -> bool
  val <= : char * char -> bool
  val > : char * char -> bool
  val >= : char * char -> bool
  val chr : int -> char
  val compare : char * char -> order
  val contains : string -> char -> bool
  val fromCString : string -> char option
  val fromString : string -> char option
  val isAlpha : char -> bool
  val isAlphaNum : char -> bool
  val isAscii : char -> bool
  val isCntrl : char -> bool
  val isDigit : char -> bool
  val isGraph : char -> bool
  val isHexDigit : char -> bool
  val isLower : char -> bool
  val isPrint : char -> bool
  val isPunct : char -> bool
  val isSpace : char -> bool
  val isUpper : char -> bool
  val maxChar : char
  val maxOrd : int
  val minChar : char
  val notContains : string -> char -> bool
  val ord : char -> int
  val pred : char -> char
  val scan : (Char.char, 'a) StringCvt.reader -> (char, 'a) StringCvt.reader
  val succ : char -> char
  val toCString : char -> string
  val toLower : char -> char
  val toString : char -> string
  val toUpper : char -> char
end
```

Structures that implement the signature

- `Char : CHAR`

26.9 COMMAND_LINE

Provide command line data for a executable program written in SML#.

```
signature COMMAND_LINE =
sig
  val arguments : unit -> string list
  val name : unit -> string
end
```

Structures that implement the signature

- `CommandLine : COMMANDLINE`

26.10 DATE

Provide date data structures and their primitives.

```
signature DATE =
sig
  datatype weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  datatype month = Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec
  type date
  exception Date
  val compare : date * date -> order
  val date : {year : int,
              month : month,
              day : int,
              hour : int,
              minute : int,
              second : int,
              offset : Time.time option} -> date
  val day : date -> int
  val fmt : string -> date -> string
  val fromString : string -> date option
  val fromTimeLocal : Time.time -> date
  val fromTimeUniv : Time.time -> date
  val hour : date -> int
  val isDst : date -> bool option
  val localOffset : unit -> Time.time
  val minute : date -> int
  val month : date -> month
  val offset : date -> Time.time option
  val scan : (char, 'a) StringCvt.reader -> (date, 'a) StringCvt.reader
  val second : date -> int
  val toString : date -> string
  val toTime : date -> Time.time
  val weekDay : date -> weekday
  val year : date -> int
  val yearDay : date -> int
end
```

Structures that implement the signature

- `Date :> DATE`

  ```
  type date (= boxed)
  ```

26.11 GENERAL

Provide primitives for reference type and exception type, and commonly used exception names.

```
signature GENERAL =
sig
  type exn = exn
  datatype order = datatype order
  eqtype unit
  exception Bind
  exception Chr
  exception Div
  exception Domain
  exception Fail of string
  exception Match
```

```
  exception Overflow
  exception Size
  exception Span
  exception Subscript
  val ! : 'a ref -> 'a
  val := : 'a ref * 'a -> unit
  val before : 'a * unit -> 'a
  val exnMessage : exn -> string
  val exnName : exn -> string
  val ignore : 'a -> unit
  val o : ('b -> 'c) * ('a -> 'b) -> 'a -> 'c
end
```

Structures that implement the signature

- `General : GENERAL`

26.12 IEEE_REAL

Provide floating-point number representations in IEEE standard.

```
signature IEEE_REAL =
sig
  type decimal_approx = {class : float_class,
                         sign : bool,
                         digits : int list,
                         exp : int}
  datatype float_class = NAN | INF | ZERO | NORMAL | SUBNORMAL
  datatype real_order = LESS | EQUAL | GREATER | UNORDERED
  datatype rounding_mode = TO_NEAREST | TO_NEGINF | TO_POSINF | TO_ZERO
  exception Unordered
  val fromString : string -> decimal_approx option
  val getRoundingMode : unit -> rounding_mode
  val scan : (char, 'a) StringCvt.reader -> (decimal_approx, 'a) StringCvt.reader
  val setRoundingMode : rounding_mode -> unit
  val toString : decimal_approx -> string
end
```

Structures that implement the signature

- `IEEEReal : IEEE_REAL`

26.13 IO

This structure defines common exceptions and data for IO processing.

```
signature IO =
sig
  exception BlockingNotSupported
  exception ClosedStream
  exception Io of {name : string, function : string, cause : exn}
  exception NonblockingNotSupported
  exception RandomAccessNotSupported
  datatype buffer_mode = NO_BUF | LINE_BUF | BLOCK_BUF
end
```

Structures that implement the signature

- `IO : IO`

26.14 INTEGER

This provides primitives for signed integers.

```
signature INTEGER =
sig
  eqtype int
  val * : int * int -> int
  val + : int * int -> int
  val - : int * int -> int
  val < : int * int -> bool
  val <= : int * int -> bool
  val > : int * int -> bool
  val >= : int * int -> bool
  val abs : int -> int
  val compare : int * int -> order
  val div : int * int -> int
  val fmt : StringCvt.radix -> int -> string
  val fromInt : Int.int -> int
  val fromLarge : LargeInt.int -> int
  val fromString : string -> int option
  val max : int * int -> int
  val maxInt : int option
  val min : int * int -> int
  val minInt : int option
  val mod : int * int -> int
  val precision : Int.int option
  val quot : int * int -> int
  val rem : int * int -> int
  val sameSign : int * int -> bool
  val scan : StringCvt.radix -> (char, 'a) StringCvt.reader -> (int, 'a) StringCvt.reader
  val sign : int -> Int.int
  val toInt : int -> Int.int
  val toLarge : int -> LargeInt.int
  val toString : int -> string
  val ~ : int -> int
end
```

INTEGER シグネチャを実装.

Structures that implement the signature

- `Int : INTEGER`

  ```
  type int = int
  ```

  `Int32` and `Position` are structure replications of `Int`.

- `Int64 : INTEGER`

  ```
  type int = int64
  ```

  `LargeInt` is a structure replication of `Int64`.

- `Int8 : INTEGER`

  ```
  type int = int8
  ```

26.15 INT_INF

Primitive

```
signature INT_INF =
sig
  include INTEGER
  val << : int * Word.word -> int
  val andb : int * int -> int
  val divMod : int * int -> int * int
  val log2 : int -> Int.int
  val notb : int -> int
  val orb : int * int -> int
  val pow : int * Int.int -> int
  val quotRem : int * int -> int * int
  val xorb : int * int -> int
  val ~>> : int * Word.word -> int
end
```

Structures that implement the signature

- `IntInf : INT_INF`

  ```
  type int = intInf
  ```

26.16 LIST

This provide primitive functions for the built-in list datatype.

```
signature LIST =
sig
  type 'a list = 'a list
  exception Empty
  val @ : 'a list * 'a list -> 'a list
  val all : ('a -> bool) -> 'a list -> bool
  val app : ('a -> unit) -> 'a list -> unit
  val collate : ('a * 'a -> order) -> 'a list * 'a list -> order
  val concat : 'a list list -> 'a list
  val drop : 'a list * int -> 'a list
  val exists : ('a -> bool) -> 'a list -> bool
  val filter : ('a -> bool) -> 'a list -> 'a list
  val find : ('a -> bool) -> 'a list -> 'a option
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
  val getItem : 'a list -> ('a * 'a list) option
  val hd : 'a list -> 'a
  val last : 'a list -> 'a
  val length : 'a list -> int
  val map : ('a -> 'b) -> 'a list -> 'b list
  val mapPartial : ('a -> 'b option) -> 'a list -> 'b list
  val nth : 'a list * int -> 'a
  val null : 'a list -> bool
  val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
  val rev : 'a list -> 'a list
  val revAppend : 'a list * 'a list -> 'a list
  val tabulate : int * (int -> 'a) -> 'a list
  val take : 'a list * int -> 'a list
  val tl : 'a list -> 'a list
end
```

Structures that implement the signature

- `List : LIST`

```
type 'a list = 'a list
```

26.17 LIST_PAIR

Provide generic functions for manipulating pair of lists.

```
signature LIST_PAIR =
sig
  exception UnequalLengths
  val all : ('a * 'b -> bool) -> 'a list * 'b list -> bool
  val allEq : ('a * 'b -> bool) -> 'a list * 'b list -> bool
  val app : ('a * 'b -> unit) -> 'a list * 'b list -> unit
  val appEq : ('a * 'b -> unit) -> 'a list * 'b list -> unit
  val exists : ('a * 'b -> bool) -> 'a list * 'b list -> bool
  val foldl : ('a * 'b * 'c -> 'c) -> 'c -> 'a list * 'b list -> 'c
  val foldlEq : ('a * 'b * 'c -> 'c) -> 'c -> 'a list * 'b list -> 'c
  val foldr : ('a * 'b * 'c -> 'c) -> 'c -> 'a list * 'b list -> 'c
  val foldrEq : ('a * 'b * 'c -> 'c) -> 'c -> 'a list * 'b list -> 'c
  val map : ('a * 'b -> 'c) -> 'a list * 'b list -> 'c list
  val mapEq : ('a * 'b -> 'c) -> 'a list * 'b list -> 'c list
  val unzip : ('a * 'b) list -> 'a list * 'b list
  val zip : 'a list * 'b list -> ('a * 'b) list
  val zipEq : 'a list * 'b list -> ('a * 'b) list
end
```

Structures that implement the signature

- `ListPair : LIST_PAIR`

26.18 MONO_ARRAY

Provide a monomorphic array type and its primitive functions.

```
signature MONO_ARRAY =
sig
  eqtype array
  type elem
  type vector
  val all : (elem -> bool) -> array -> bool
  val app : (elem -> unit) -> array -> unit
  val appi : (int * elem -> unit) -> array -> unit
  val array : int * elem -> array
  val collate : (elem * elem -> order) -> array * array -> order
  val copy : {src : array, dst : array, di : int} -> unit
  val copyVec : {src : vector, dst : array, di : int} -> unit
  val exists : (elem -> bool) -> array -> bool
  val find : (elem -> bool) -> array -> elem option
  val findi : (int * elem -> bool) -> array -> (int * elem) option
  val foldl : (elem * 'b -> 'b) -> 'b -> array -> 'b
  val foldli : (int * elem * 'b -> 'b) -> 'b -> array -> 'b
  val foldr : (elem * 'b -> 'b) -> 'b -> array -> 'b
  val foldri : (int * elem * 'b -> 'b) -> 'b -> array -> 'b
  val fromList : elem list -> array
  val length : array -> int
  val maxLen : int
```

```
  val modify : (elem -> elem) -> array -> unit
  val modifyi : (int * elem -> elem) -> array -> unit
  val sub : array * int -> elem
  val tabulate : int * (int -> elem) -> array
  val update : array * int * elem -> unit
  val vector : array -> vector
end
```

Structures that implement the signature

- `CharArray : MONO_ARRAY`

  ```
  type array = char array
  type elem = char
  type vector = string
  ```

- `Word8Array : MONO_ARRAY`

  ```
  type array = word8 array
  type elem = word8
  type vector = word8 vector
  ```

26.19 MONO_ARRAY_SLICE

Provide primitives for manipulating slices (sub-arrays) of monomorpic arrays.

```
signature MONO_ARRAY_SLICE =
sig
  type array
  type elem
  type slice
  type vector
  type vector_slice
  val all : (elem -> bool) -> slice -> bool
  val app : (elem -> unit) -> slice -> unit
  val appi : (int * elem -> unit) -> slice -> unit
  val base : slice -> array * int * int
  val collate : (elem * elem -> order) -> slice * slice -> order
  val copy : {src : slice, dst : array, di : int} -> unit
  val copyVec : {src : vector_slice, dst : array, di : int} -> unit
  val exists : (elem -> bool) -> slice -> bool
  val find : (elem -> bool) -> slice -> elem option
  val findi : (int * elem -> bool) -> slice -> (int * elem) option
  val foldl : (elem * 'b -> 'b) -> 'b -> slice -> 'b
  val foldli : (int * elem * 'b -> 'b) -> 'b -> slice -> 'b
  val foldr : (elem * 'b -> 'b) -> 'b -> slice -> 'b
  val foldri : (int * elem * 'b -> 'b) -> 'b -> slice -> 'b
  val full : array -> slice
  val getItem : slice -> (elem * slice) option
  val isEmpty : slice -> bool
  val length : slice -> int
  val modify : (elem -> elem) -> slice -> unit
  val modifyi : (int * elem -> elem) -> slice -> unit
  val slice : array * int * int option -> slice
  val sub : slice * int -> elem
  val subslice : slice * int * int option -> slice
  val update : slice * int * elem -> unit
  val vector : slice -> vector
end
```

Structures that implement the signature

- CharArraySlice :> MONO_ARRAY_SLICE

```
type array = char array
type elem = char
type slice (= boxed)
type vector = string
type vector_slice = CharVectorSlice.slice
```

- Word8ArraySlice :> MONO_ARRAY_SLICE

```
type array = word8 array
type elem = word8
type slice (= boxed)
type vector = word8 vector
type vector_slice = Word8VectorSlice.slice
```

26.20 MONO_VECTOR

Provide a monomorphic vector type and its primitive functions.

```
signature MONO_VECTOR =
sig
  type vector
  type elem
  val all : (elem -> bool) -> vector -> bool
  val app : (elem -> unit) -> vector -> unit
  val appi : (int * elem -> unit) -> vector -> unit
  val collate : (elem * elem -> order) -> vector * vector -> order
  val concat : vector list -> vector
  val exists : (elem -> bool) -> vector -> bool
  val find : (elem -> bool) -> vector -> elem option
  val findi : (int * elem -> bool) -> vector -> (int * elem) option
  val foldl : (elem * 'a -> 'a) -> 'a -> vector -> 'a
  val foldli : (int * elem * 'a -> 'a) -> 'a -> vector -> 'a
  val foldr : (elem * 'a -> 'a) -> 'a -> vector -> 'a
  val foldri : (int * elem * 'a -> 'a) -> 'a -> vector -> 'a
  val fromList : elem list -> vector
  val length : vector -> int
  val map : (elem -> elem) -> vector -> vector
  val mapi : (int * elem -> elem) -> vector -> vector
  val maxLen : int
  val sub : vector * int -> elem
  val tabulate : int * (int -> elem) -> vector
  val update : vector * int * elem -> vector
end
```

Structures that implement the signature

- CharVector : MONO_VECTOR

```
type elem = char
type vector = string
```

- Word8Array : MONO_ARRAY

```
type elem = word8
type vector = word8 vector
```

26.21 MONO_VECTOR_SLICE

Provide primitives for manipulating slices (sub-vectors) of monomorpic vectors.

```
signature MONO_VECTOR_SLICE =
sig
  type elem
  type slice
  type vector
  val all : (elem -> bool) -> slice -> bool
  val app : (elem -> unit) -> slice -> unit
  val appi : (int * elem -> unit) -> slice -> unit
  val base : slice -> vector * int * int
  val collate : (elem * elem -> order) -> slice * slice -> order
  val concat : slice list -> vector
  val exists : (elem -> bool) -> slice -> bool
  val find : (elem -> bool) -> slice -> elem option
  val findi : (int * elem -> bool) -> slice -> (int * elem) option
  val foldl : (elem * 'b -> 'b) -> 'b -> slice -> 'b
  val foldli : (int * elem * 'b -> 'b) -> 'b -> slice -> 'b
  val foldr : (elem * 'b -> 'b) -> 'b -> slice -> 'b
  val foldri : (int * elem * 'b -> 'b) -> 'b -> slice -> 'b
  val full : vector -> slice
  val getItem : slice -> (elem * slice) option
  val isEmpty : slice -> bool
  val length : slice -> int
  val map : (elem -> elem) -> slice -> vector
  val mapi : (int * elem -> elem) -> slice -> vector
  val slice : vector * int * int option -> slice
  val sub : slice * int -> elem
  val subslice : slice * int * int option -> slice
  val vector : slice -> vector
end
```

Structures that implement the signature

- `CharVectorSlice :> MONO_VECTOR_SLICE`

  ```
  type elem = char
  type slice (= boxed)
  type vector = string
  ```

- `Word8VectorSlice :> MONO_VECTOR_SLICE`

  ```
  type elem = word8
  type slice (= boxed)
  type vector = word8 vector
  ```

26.22 OPTION

Provide an operation type and primitive functions.

```
signature OPTION =
sig
  datatype 'a option = NONE | SOME of 'a
  exception Option
  val app : ('a -> unit) -> 'a option -> unit
  val compose : ('a -> 'b) * ('c -> 'a option) -> 'c -> 'b option
  val composePartial : ('a -> 'b option) * ('c -> 'a option) -> 'c -> 'b option
  val filter : ('a -> bool) -> 'a -> 'a option
```

```
  val getOpt : 'a option * 'a -> 'a
  val isSome : 'a option -> bool
  val join : 'a option option -> 'a option
  val map : ('a -> 'b) -> 'a option -> 'b option
  val mapPartial : ('a -> 'b option) -> 'a option -> 'b option
  val valOf : 'a option -> 'a
end
```

- Option : OPTION

26.23 OS

Provide interface functions to the underling operating system.

```
signature OS =
sig
  structure FileSys : OS_FILE_SYS
  structure IO : OS_IO
  structure Path : OS_PATH
  structure Process : OS_PROCESS
  eqtype syserror
  exception SysErr of string * syserror option
  val errorMsg : syserror -> string
  val errorName : syserror -> string
  val syserror : string -> syserror option
end
```

Nested signatures

- OS_FILE_SYS (26.24)
- OS_IO (26.25)
- OS_PATH (26.26)
- OS_PROCESS (26.27)

Structures that implement the signature

- OS : OS

```
eqtype syserror (= int)
structure FileSys = struct
  type dirstream (= boxed)
end
structure Process = struct
  type status = int
end
structure IO = struct
  type iodesc (= int)
  eqtype iodesc_kind (= word)
end
```

26.24 OS_FILE_SYS

Provide interface primitives for the underlying OS file system.

```
signature OS_FILE_SYS =
sig
  datatype access_mode = A_READ | A_WRITE | A_EXEC
```

```
  type dirstream
  eqtype file_id
  val access : string * access_mode list -> bool
  val chDir : string -> unit
  val closeDir : dirstream -> unit
  val compare : file_id * file_id -> order
  val fileId : string -> file_id
  val fileSize : string -> Position.int
  val fullPath : string -> string
  val getDir : unit -> string
  val hash : file_id -> word
  val isDir : string -> bool
  val isLink : string -> bool
  val mkDir : string -> unit
  val modTime : string -> Time.time
  val openDir : string -> dirstream
  val readDir : dirstream -> string option
  val readLink : string -> string
  val realPath : string -> string
  val remove : string -> unit
  val rename : {old : string, new : string} -> unit
  val rewindDir : dirstream -> unit
  val rmDir : string -> unit
  val setTime : string * Time.time option -> unit
  val tmpName : unit -> string
end
```

26.25 OS_IO

Provide interface primitives for the underlying OS IO.

```
signature OS_IO =
sig
  eqtype iodesc
  eqtype iodesc_kind
  eqtype poll_desc
  type poll_info
  exception Poll
  structure Kind : sig
    val device : iodesc_kind
    val dir : iodesc_kind
    val file : iodesc_kind
    val pipe : iodesc_kind
    val socket : iodesc_kind
    val symlink : iodesc_kind
    val tty : iodesc_kind
  end
  val compare : iodesc * iodesc -> order
  val hash : iodesc -> word
  val infoToPollDesc : poll_info -> poll_desc
  val isIn : poll_info -> bool
  val isOut : poll_info -> bool
  val isPri : poll_info -> bool
  val kind : iodesc -> iodesc_kind
  val poll : poll_desc list * Time.time option -> poll_info list
  val pollDesc : iodesc -> poll_desc option
  val pollIn : poll_desc -> poll_desc
  val pollOut : poll_desc -> poll_desc
```

```
  val pollPri : poll_desc -> poll_desc
  val pollToIODesc : poll_desc -> iodesc
end
```

26.26 OS_PATH

Provide primitives for manipulating file path structures in the underlying OS IO.

```
signature OS_PATH =
sig
  exception InvalidArc
  exception Path
  val base : string -> string
  val concat : string * string -> string
  val currentArc : string
  val dir : string -> string
  val ext : string -> string option
  val file : string -> string
  val fromString : string -> {isAbs : bool, vol : string, arcs : string list}
  val fromUnixPath : string -> string
  val getParent : string -> string
  val getVolume : string -> string
  val isAbsolute : string -> bool
  val isCanonical : string -> bool
  val isRelative : string -> bool
  val isRoot : string -> bool
  val joinBaseExt : {base : string, ext : string option} -> string
  val joinDirFile : {dir : string, file : string} -> string
  val mkAbsolute : {path : string, relativeTo : string} -> string
  val mkCanonical : string -> string
  val mkRelative : {path : string, relativeTo : string} -> string
  val parentArc : string
  val splitBaseExt : string -> {base : string, ext : string option}
  val splitDirFile : string -> {dir : string, file : string}
  val toString : {isAbs : bool, vol : string, arcs : string list} -> string
  val toUnixPath : string -> string
  val validVolume : {isAbs : bool, vol : string} -> bool
end
```

26.27 OS_PROCESS

Provide primitives for manipulating processes and threads in the underlying OS IO.

```
signature OS_PROCESS =
sig
  type status
  val atExit : (unit -> unit) -> unit
  val exit : status -> 'a
  val failure : status
  val getEnv : string -> string option
  val isSuccess : status -> bool
  val sleep : Time.time -> unit
  val success : status
  val system : string -> status
  val terminate : status -> 'a
end
```

26.28 REAL

Provide primitive functions for floating point numbers.

```
signature REAL =
sig
  type real
  structure Math : MATH where type real = real
  val != : real * real -> bool
  val * : real * real -> real
  val *+ : real * real * real -> real
  val *- : real * real * real -> real
  val + : real * real -> real
  val - : real * real -> real
  val / : real * real -> real
  val < : real * real -> bool
  val <= : real * real -> bool
  val == : real * real -> bool
  val > : real * real -> bool
  val >= : real * real -> bool
  val ?= : real * real -> bool
  val abs : real -> real
  val ceil : real -> int
  val checkFloat : real -> real
  val class : real -> IEEEReal.float_class
  val compare : real * real -> order
  val compareReal : real * real -> IEEEReal.real_order
  val copySign : real * real -> real
  val floor : real -> int
  val fmt : StringCvt.realfmt -> real -> string
  val fromDecimal : IEEEReal.decimal_approx -> real option
  val fromInt : int -> real
  val fromLarge : IEEEReal.rounding_mode -> LargeReal.real -> real
  val fromLargeInt : LargeInt.int -> real
  val fromManExp : {man : real, exp : int} -> real
  val fromString : string -> real option
  val isFinite : real -> bool
  val isNan : real -> bool
  val isNormal : real -> bool
  val max : real * real -> real
  val maxFinite : real
  val min : real * real -> real
  val minNormalPos : real
  val minPos : real
  val negInf : real
  val nextAfter : real * real -> real
  val posInf : real
  val precision : int
  val radix : int
  val realCeil : real -> real
  val realFloor : real -> real
  val realMod : real -> real
  val realRound : real -> real
  val realTrunc : real -> real
  val rem : real * real -> real
  val round : real -> int
  val sameSign : real * real -> bool
  val scan : (char, 'a) StringCvt.reader -> (real, 'a) StringCvt.reader
  val sign : real -> int
```

```
  val signBit : real -> bool
  val split : real -> {whole : real, frac : real}
  val toDecimal : real -> IEEEReal.decimal_approx
  val toInt : IEEEReal.rounding_mode -> real -> int
  val toLarge : real -> LargeReal.real
  val toLargeInt : IEEEReal.rounding_mode -> real -> LargeInt.int
  val toManExp : real -> {man : real, exp : int}
  val toString : real -> string
  val trunc : real -> int
  val unordered : real * real -> bool
  val ~ : real -> real
end
```

Nested signatures

- MATH (26.29)

Structures that implement the signature

- Real : REAL

  ```
  type real = real
  ```

  Real64 and LargeReal are structure replications of Real.

- Real32 : REAL

  ```
  type real = real32
  ```

26.29 MATH

Provide mathematical functions on floating-point numbers. This is a sub-signature of REAL.

```
signature MATH =
sig
  type real
  val acos : real -> real
  val asin : real -> real
  val atan : real -> real
  val atan2 : real * real -> real
  val cos : real -> real
  val cosh : real -> real
  val e : real
  val exp : real -> real
  val ln : real -> real
  val log10 : real -> real
  val pi : real
  val pow : real * real -> real
  val sin : real -> real
  val sinh : real -> real
  val sqrt : real -> real
  val tan : real -> real
  val tanh : real -> real
end
```

26.30 STRING

Provide primitives for manipulating character strings.

```
signature STRING =
sig
  eqtype char
  eqtype string
  val < : string * string -> bool
  val <= : string * string -> bool
  val > : string * string -> bool
  val >= : string * string -> bool
  val ^ : string * string -> string
  val collate : (char * char -> order) -> string * string -> order
  val compare : string * string -> order
  val concat : string list -> string
  val concatWith : string -> string list -> string
  val explode : string -> char list
  val extract : string * int * int option -> string
  val fields : (char -> bool) -> string -> string list
  val fromCString : string -> string option
  val fromString : string -> string option
  val implode : char list -> string
  val isPrefix : string -> string -> bool
  val isSubstring : string -> string -> bool
  val isSuffix : string -> string -> bool
  val map : (char -> char) -> string -> string
  val maxSize : int
  val scan : (char, 'a) StringCvt.reader -> (string, 'a) StringCvt.reader
  val size : string -> int
  val str : char -> string
  val sub : string * int -> char
  val substring : string * int * int -> string
  val toCString : string -> string
  val toString : string -> string
  val tokens : (char -> bool) -> string -> string list
  val translate : (char -> string) -> string -> string
end
```

Structures that implement the signature

- `String : STRING`

  ```
  type char = char
  type string = string
  ```

26.31 STRING_CVT

Provide primitives for string manipulation.

```
signature STRING_CVT =
sig
  datatype radix = BIN | OCT | DEC | HEX
  type ('a,'b) reader = 'b -> ('a * 'b) option
  datatype realfmt =
      SCI of int option
    | FIX of int option
    | GEN of int option
    | EXACT
  type cs
  val dropl : (char -> bool) -> (char, 'a) reader -> 'a -> 'a
  val padLeft : char -> int -> string -> string
```

```
  val padRight : char -> int -> string -> string
  val scanString : ((char, cs) reader -> ('a, cs) reader) -> string -> 'a option
  val skipWS : (char, 'a) reader -> 'a -> 'a
  val splitl : (char -> bool) -> (char, 'a) reader -> 'a -> string * 'a
  val takel : (char -> bool) -> (char, 'a) reader -> 'a -> string
end
```

Structures that implement the signature

- `StringCvt : STRING_CVT`

  ```
  type cs (= boxed)
  ```

26.32 SUBSTRING

Provide primitives for a sub-string type and its primitive functions.

```
signature SUBSTRING =
sig
  eqtype char
  eqtype string
  type substring
  val app : (char -> unit) -> substring -> unit
  val base : substring -> string * int * int
  val collate : (char * char -> order) -> substring * substring -> order
  val compare : substring * substring -> order
  val concat : substring list -> string
  val concatWith : string -> substring list -> string
  val dropl : (char -> bool) -> substring -> substring
  val dropr : (char -> bool) -> substring -> substring
  val explode : substring -> char list
  val extract : string * int * int option -> substring
  val fields : (char -> bool) -> substring -> substring list
  val first : substring -> char option
  val foldl : (char * 'a -> 'a) -> 'a -> substring -> 'a
  val foldr : (char * 'a -> 'a) -> 'a -> substring -> 'a
  val full : string -> substring
  val getc : substring -> (char * substring) option
  val isEmpty : substring -> bool
  val isPrefix : string -> substring -> bool
  val isSubstring : string -> substring -> bool
  val isSuffix : string -> substring -> bool
  val position : string -> substring -> substring * substring
  val size : substring -> int
  val slice : substring * int * int option -> substring
  val span : substring * substring -> substring
  val splitAt : substring * int -> substring * substring
  val splitl : (char -> bool) -> substring -> substring * substring
  val splitr : (char -> bool) -> substring -> substring * substring
  val string : substring -> string
  val sub : substring * int -> char
  val substring : string * int * int -> substring
  val takel : (char -> bool) -> substring -> substring
  val taker : (char -> bool) -> substring -> substring
  val tokens : (char -> bool) -> substring -> substring list
  val translate : (char -> string) -> substring -> string
  val triml : int -> substring -> substring
  val trimr : int -> substring -> substring
end
```

Structures that implement the signature

- ```
  Substring :> SUBSTRING
    where type substring = CharVectorSlice.slice
    where type string = string
    where type char = char
  ```

26.33 TEXT

Provide text structures.

```
signature TEXT =
sig
  structure Char : CHAR
  structure CharArray : MONO_ARRAY
  structure CharArraySlice : MONO_ARRAY_SLICE
  structure CharVector : MONO_VECTOR
  structure CharVectorSlice : MONO_VECTOR_SLICE
  structure String : STRING
  structure Substring : SUBSTRING
  sharing type
      Char.char
    = String.char
    = Substring.char
    = CharVector.elem
    = CharArray.elem
    = CharVectorSlice.elem
    = CharArraySlice.elem
  sharing type
      Char.string
    = String.string
    = Substring.string
    = CharVector.vector
    = CharArray.vector
    = CharVectorSlice.vector
    = CharArraySlice.vector
  sharing type
      CharArray.array
    = CharArraySlice.array
  sharing type
      CharVectorSlice.slice
    = CharArraySlice.vector_slice
end
```

Structures that implement the signature

- `Text : TEXT`

26.34 TEXT_IO

```
signature TEXT_IO =
sig
  structure StreamIO : TEXT_STREAM_IO
    where type reader = TextPrimIO.reader
    where type writer = TextPrimIO.writer
    where type pos = TextPrimIO.pos
  type elem = StreamIO.elem
  type instream
```
```

```
  type outstream
  type vector = StreamIO.vector
  val canInput : instream * int -> int option
  val closeIn : instream -> unit
  val closeOut : outstream -> unit
  val endOfStream : instream -> bool
  val flushOut : outstream -> unit
  val getInstream : instream -> StreamIO.instream
  val getOutstream : outstream -> StreamIO.outstream
  val getPosOut : outstream -> StreamIO.out_pos
  val input : instream -> vector
  val input1 : instream -> elem option
  val inputAll : instream -> vector
  val inputLine : instream -> string option
  val inputN : instream * int -> vector
  val lookahead : instream -> elem option
  val mkInstream : StreamIO.instream -> instream
  val mkOutstream : StreamIO.outstream -> outstream
  val openAppend : string -> outstream
  val openIn : string -> instream
  val openOut : string -> outstream
  val openString : string -> instream
  val output : outstream * vector -> unit
  val output1 : outstream * elem -> unit
  val outputSubstr : outstream * substring -> unit
  val print : string -> unit
  val scanStream
     : ((Char.char, StreamIO.instream) StringCvt.reader
 -> ('a, StreamIO.instream) StringCvt.reader)
 -> instream
 -> 'a option
  val setInstream : instream * StreamIO.instream -> unit
  val setOutstream : outstream * StreamIO.outstream -> unit
  val setPosOut : outstream * StreamIO.out_pos -> unit
  val stdErr : outstream
  val stdIn : instream
  val stdOut : outstream
end
```

Nested signatures

- `TEXT_STREAM_IO` (26.35)

Structures that implement the signature

- `TextIO : TEXT_IO`

26.35 TEXT_STREAM_IO

```
signature TEXT_STREAM_IO =
sig
  include STREAM_IO
    where type vector = CharVector.vector
    where type elem = Char.char
  val inputLine : instream -> (string * instream) option
  val outputSubstr : outstream * substring -> unit
end
```

26.36 PRIM_IO

Provide low-level IO primitives.

```
signature PRIM_IO =
sig
  type array
  type array_slice
  type elem
  eqtype pos
  datatype reader =
      RD of {name : string,
             chunkSize : int,
             readVec : (int -> vector) option,
             readArr : (array_slice -> int) option,
             readVecNB : (int -> vector option) option,
             readArrNB : (array_slice -> int option) option,
             block : (unit -> unit) option,
             canInput : (unit -> bool) option,
             avail : unit -> int option,
             getPos : (unit -> pos) option,
             setPos : (pos -> unit) option,
             endPos : (unit -> pos) option,
             verifyPos : (unit -> pos) option,
             close : unit -> unit,
             ioDesc : OS.IO.iodesc option}
  type vector
  type vector_slice
  datatype writer =
      WR of {name : string,
             chunkSize : int,
             writeVec : (vector_slice -> int) option,
             writeArr : (array_slice -> int) option,
             writeVecNB : (vector_slice -> int option) option,
             writeArrNB : (array_slice -> int option) option,
             block : (unit -> unit) option,
             canOutput : (unit -> bool) option,
             getPos : (unit -> pos) option,
             setPos : (pos -> unit) option,
             endPos : (unit -> pos) option,
             verifyPos : (unit -> pos) option,
             close : unit -> unit,
             ioDesc : OS.IO.iodesc option}
  val augmentReader : reader -> reader
  val augmentWriter : writer -> writer
  val compare : pos * pos -> order
  val nullRd : unit -> reader
  val nullWr : unit -> writer
  val openVector : vector -> reader
end
```

Structures that implement the signature

- `structure BinPrimIO :> PRIM_IO`
  `  where type array = Word8Array.array`
  `  where type vector = Word8Vector.vector`
  `  where type elem = Word8.word`
  `  where type pos = Position.int`

- structure TextPrimIO :> PRIM_IO
    where type array = CharArray.array
    where type vector = CharVector.vector
    where type elem = Char.char

26.37 TIME

Provide a datatype for time and its primitive operations.

```
signature TIME =
sig
  eqtype time
  exception Time
  val + : time * time -> time
  val - : time * time -> time
  val < : time * time -> bool
  val <= : time * time -> bool
  val > : time * time -> bool
  val >= : time * time -> bool
  val compare : time * time -> order
  val fmt : int -> time -> string
  val fromMicroseconds : LargeInt.int -> time
  val fromMilliseconds : LargeInt.int -> time
  val fromNanoseconds : LargeInt.int -> time
  val fromReal : LargeReal.real -> time
  val fromSeconds : LargeInt.int -> time
  val fromString : string -> time option
  val now : unit -> time
  val scan : (char, 'a) StringCvt.reader -> (time, 'a) StringCvt.reader
  val toMicroseconds : time -> LargeInt.int
  val toMilliseconds : time -> LargeInt.int
  val toNanoseconds : time -> LargeInt.int
  val toReal : time -> LargeReal.real
  val toSeconds : time -> LargeInt.int
  val toString : time -> string
  val zeroTime : time
end
```

Structures that implement the signature

- Time :> TIME

      type time (= real)

26.38 TIMER

Provide a type for time and its primitive functions.

```
signature TIMER =
sig
  type cpu_timer
  type real_timer
  val checkCPUTimer : cpu_timer -> {usr : Time.time, sys : Time.time}
  val checkCPUTimes
    : cpu_timer
 -> {nongc : {usr : Time.time, sys : Time.time}, gc : {usr : Time.time, sys : Time.time}}
  val checkGCTime : cpu_timer -> Time.time
  val checkRealTimer : real_timer -> Time.time
```

```
  val startCPUTimer : unit -> cpu_timer
  val startRealTimer : unit -> real_timer
  val totalCPUTimer : unit -> cpu_timer
  val totalRealTimer : unit -> real_timer
end
```

Structures that implement the signature

- `Timer :> TIMERE`

  ```
  type cpu_timer (= boxed)
  type real_timer (= boxed)
  ```

26.39 VECTOR

Provide a vector (immutable array) type and its primitive functions.

```
signature VECTOR =
sig
  type 'a vector = 'a Vector.vector
  val all : ('a -> bool) -> 'a vector -> bool
  val app : ('a -> unit) -> 'a vector -> unit
  val appi : (int * 'a -> unit) -> 'a vector -> unit
  val collate : ('a * 'a -> order) -> 'a vector * 'a vector -> order
  val concat : 'a vector list -> 'a vector
  val exists : ('a -> bool) -> 'a vector -> bool
  val find : ('a -> bool) -> 'a vector -> 'a option
  val findi : (int * 'a -> bool) -> 'a vector -> (int * 'a) option
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a vector -> 'b
  val foldli : (int * 'a * 'b -> 'b) -> 'b -> 'a vector -> 'b
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a vector -> 'b
  val foldri : (int * 'a * 'b -> 'b) -> 'b -> 'a vector -> 'b
  val fromList : 'a list -> 'a vector
  val length : 'a vector -> int
  val map : ('a -> 'b) -> 'a vector -> 'b vector
  val mapi : (int * 'a -> 'b) -> 'a vector -> 'b vector
  val maxLen : int
  val sub : 'a vector * int -> 'a
  val tabulate : int * (int -> 'a) -> 'a vector
  val update : 'a vector * int * 'a -> 'a vector
end
```

Structures that implement the signature

- `Vector : VECTOR`

26.40 VECTOR_SLICE

Provide a type for vectors and its primitive functions.

```
signature VECTOR_SLICE =
sig
  type 'a slice
  val all : ('a -> bool) -> 'a slice -> bool
  val app : ('a -> unit) -> 'a slice -> unit
  val appi : (int * 'a -> unit) -> 'a slice -> unit
  val base : 'a slice -> 'a Vector.vector * int * int
```

```
  val collate : ('a * 'a -> order) -> 'a slice * 'a slice -> order
  val concat : 'a slice list -> 'a Vector.vector
  val exists : ('a -> bool) -> 'a slice -> bool
  val find : ('a -> bool) -> 'a slice -> 'a option
  val findi : (int * 'a -> bool) -> 'a slice -> (int * 'a) option
  val foldl : ('a * 'b -> 'b) -> 'b -> 'a slice -> 'b
  val foldli : (int * 'a * 'b -> 'b) -> 'b -> 'a slice -> 'b
  val foldr : ('a * 'b -> 'b) -> 'b -> 'a slice -> 'b
  val foldri : (int * 'a * 'b -> 'b) -> 'b -> 'a slice -> 'b
  val full : 'a Vector.vector -> 'a slice
  val getItem : 'a slice -> ('a * 'a slice) option
  val isEmpty : 'a slice -> bool
  val length : 'a slice -> int
  val map : ('a -> 'b) -> 'a slice -> 'b Vector.vector
  val mapi : (int * 'a -> 'b) -> 'a slice -> 'b Vector.vector
  val slice : 'a Vector.vector * int * int option -> 'a slice
  val sub : 'a slice * int -> 'a
  val subslice : 'a slice * int * int option -> 'a slice
  val vector : 'a slice -> 'a Vector.vector
end
```

Structures that implement the signature

- `VectorSlice :> VECTOR_SLICE`

  ```
  type 'a slice (= boxed)
  ```

26.41 WORD

Provide primitive functions for unsigned integers.

```
signature WORD =
sig
  eqtype word
  val * : word * word -> word
  val + : word * word -> word
  val - : word * word -> word
  val < : word * word -> bool
  val << : word * Word.word -> word
  val <= : word * word -> bool
  val > : word * word -> bool
  val >= : word * word -> bool
  val >> : word * Word.word -> word
  val andb : word * word -> word
  val compare : word * word -> order
  val div : word * word -> word
  val fmt : StringCvt.radix -> word -> string
  val fromInt : int -> word
  val fromLarge : LargeWord.word -> word
  val fromLargeInt : LargeInt.int -> word
  val fromLargeWord : LargeWord.word -> word
  val fromString : string -> word option
  val max : word * word -> word
  val min : word * word -> word
  val mod : word * word -> word
  val notb : word -> word
  val orb : word * word -> word
  val scan : StringCvt.radix -> (char, 'a) StringCvt.reader -> (word, 'a) StringCvt.reader
```

```
  val toInt : word -> int
  val toIntX : word -> int
  val toLarge : word -> LargeWord.word
  val toLargeInt : word -> LargeInt.int
  val toLargeIntX : word -> LargeInt.int
  val toLargeWord : word -> LargeWord.word
  val toLargeWordX : word -> LargeWord.word
  val toLargeX : word -> LargeWord.word
  val toString : word -> string
  val wordSize : int
  val xorb : word * word -> word
  val ~ : word -> word
  val ~>> : word * Word.word -> word
end
```

Structures that implement the signature

- `Word : WORD`

  ```
  type word = word
  ```

  `Word32` is a structure replication of `Word`

- `Word64 : WORD`

  ```
  type word = word64
  ```

  `LargeWord` is a structure replication of `Word64`

- `Word8 : WORD`

  ```
  type word = word8
  ```

26.42 The top-level environment

Commonly used functions defined in libraries are replicated in the top-level environment. This section shows the top-level bindings of identifiers defined in the Standard ML Basis Library.

- infix declarations

  ```
  infix 7 * / div mod
  infix 6 + - ^
  infixr 5 :: @
  infix 4 = <> > >= < <=
  infix 3 := o
  infix 0 before
  ```

- type declarations

  ```
  type substring = Substring.substring
  datatype order = datatype General.order
  ```

- exception declarations

  ```
  exception Bind = General.Bind
  exception Chr= General.Chr
  exception Div= General.Div
  exception Domain= General.Domain
  exception Empty = List.Empty
  exception Fail = General.Fail
  exception Match= General.Match
  ```

```
exception Overflow= General.Overflow
exception Size= General.Size
exception Span= General.Span
exception Subscript = General.Subscript
exception Option = Option.Option
exception Span = General.Span
```

- val declaration

```
val = = <builtin> : [''a. ''a * ''a -> bool]}
val <> = <builtin> : [''a. ''a * ''a -> bool]}
val ! = General.!
val := = General.:=
val @ = List.@
val ^ = String.^
val app = List.app
val before = General.before
val ceil = Real.ceil
val chr = Char.chr
val concat = String.concat
val exnMessage = General.exnMessage
val exnName = General.exnName
val explode = String.explode
val floor = Real.floor
val foldl = List.foldl
val foldr = List.foldr
val getOpt = Option.getOpt
val hd = List.hd
val ignore = General.ignore
val implode = String.implode
val isSome = Option.isSome
val length = List.length
val map = List.map
val not = Bool.not
val null = List.null
val o = General.o
val ord = Char.ord
val print = TextIO.print
val real = Real.fromInt
val rev = List.rev
val round = Real.round
val size = String.size
val str = String.str
val substring = String.substring
val tl = List.tl
val trunc = Real.trunc
val valOf = Option.valOf
val vector = Vector.fromList
```

The equality check primitives = and <> are built-in functions that are directly supported by the SML# compiler.

- Overloaded identifiers

```
val * : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32}. 'a * '
val + : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32}. 'a * '
val - : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32}. 'a * '
val / : ['a::{real, real32}. 'a * 'a -> 'a]
val < : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32, string,
```

```
val <= : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32, string
val > : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32, string,
val >= : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32, string
val \ : ['a::{real, real32}. 'a -> 'a]
val abs : ['a::{int, int8, int16, int64, real, real32}. 'a -> 'a]
val div : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf}. 'a * 'a -> 'a]
val mod : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf}. 'a * 'a -> 'a]
```

Chapter 27

SML# System Library

In addition to Standard ML Basis Library, SML# provides its own libraries for exploiting SML# language features. Being SML# specific libraries, they are directly provided through interface files (smi files) without signature specification. The are organized into library interface files (smi files), which are referenced by file name (⟨*librarySmiFilePath*⟩) without specifying file paths (⟨*smiFilePath*⟩).

The current version provide the following.

Library name	structure name (section)
`"ffi.smi"`	`DynamicLink` (27.1)
	`Pointer` (27.2)
`"sql.smi"`	`SQL` (27.3)
	`SQL.Op` (27.4)
	`SQL.Numeric` (27.5)
`"thread.smi"`	`Pthread` (27.6)
	`Myth` (27.7)
`"reify.smi"`	`Dynamic` (27.8)

These libraries may be referred by the compiler to realize the SML# extensions. To use the following SML# extensions in separate compilation mode, you must `_require` the following libraries from your interface file:

- To use SQL expressions, you must `_require "sql.smi"`.
- To call a polymorphic function with `#reify` kind, you must `_require "reify.smi"`.

27.1 DynamicLink

Interfce

```
structure DynamicLink =
  struct
    type lib (= boxed)
    datatype mode = LAZY | NOW
    datatype scope = GLOBAL | LOCAL
    val default : unit -> lib
    val dlclose : lib -> unit
    val dlopen  : string -> lib
    val dlopen' : string * scope * mode -> lib
    val dlsym : lib * string -> codeptr
    val dlsym' : lib * string -> unit ptr
    val next : unit -> lib
  end
```

Types

- `lib` An abstract type representing an internal handle of opened dynamic link library.
- `mode` Open mode for `dlopen'` primitive. `NOW` indicates that the library file is opened when `dlopen'` is called. `LAZY` indicate that `dlopen'` checks and only prepares to open the library file, which will be opened when some reference will be made at runtime.

27.2 Pointer

Interfce

```
structure Pointer =
  struct
    val NULL : ['a. unit -> 'a ptr]
    val advance : ['a. 'a ptr * int -> 'a ptr]
    val importBytes : word8 ptr * int -> word8 vector
    val importString : char ptr -> string
    val isNull : ['a. 'a ptr -> bool]
    val load : ['a. 'a ptr -> 'a]
    val store : ['a. 'a ptr * 'a -> unit]
  end
```

27.3 SQL

See Section 22.8 for details of the SQL library.

Nested structures

- `Op`(27.4)
- `Numeric`(27.5)

27.4 SQL.Op

See Section 22.9 for details of the SQL.Op structure.

27.5 SQL.Numeric

See Section 22.10 for details of the SQL.Num structure.

27.6 Pthread

Interfce

```
structure Pthread =
  struct
    type thread (= *)
    structure Thread =
      struct
        type thread = thread
        val create : (unit -> int) -> thread
        val detach : thread -> unit
        val join : thread -> int
        val exit : int -> unit
        val self : unit -> thread
        val equal : thread * thread -> bool
      end
```

```
    type mutex (= array)
    structure Mutex =
      struct
        type mutex = mutex
        val create : unit -> mutex
        val lock : mutex -> unit
        val unlock : mutex -> unit
        val trylock : mutex -> bool
        val destroy : mutex -> unit
      end
    type cond (= array)
    structure Cond =
      struct
        type cond = cond
        val create : unit -> cond
        val signal : cond -> unit
        val broadcast : cond -> unit
        val wait : cond * mutex -> unit
        val destroy : cond -> unit
      end
  end
```

27.7 Myth

Interfce

```
structure Myth =
  struct
    type thread (= *)
    structure Thread =
      struct
        type thread = thread
        val create : (unit -> int) -> thread
        val detach : thread -> unit
        val join : thread -> int
        val exit : int -> unit
        val yield : unit -> unit
        val self : unit -> thread
        val equal : thread * thread -> bool
      end
    type mutex (= array)
    structure Mutex =
      struct
        type mutex = mutex
        val create : unit -> mutex
        val lock : mutex -> unit
        val unlock : mutex -> unit
        val trylock : mutex -> bool
        val destroy : mutex -> unit
      end
    type cond (= array)
    structure Cond =
      struct
        type cond = cond
        val create : unit -> cond
        val signal : cond -> unit
        val broadcast : cond -> unit
        val wait : cond * mutex -> unit
```

```
      val destroy : cond -> unit
    end
  type barrier (= array)
  structure Barrier =
    struct
      type barrier = barrier
      val create : int -> barrier
      val wait : barrier -> bool
      val destroy : barrier -> unit
    end
end
```

27.8 Dynamic

Interfce

```
structure Dynamic =
  struct
    datatype term =
        ARRAY of ty * boxed
      | ARRAY_PRINT of term array
      | BOOL of bool
      | BOXED of boxed
      | BOUNDVAR
      | BUILTIN
      | CHAR of char
      | CODEPTR of word64
      | DATATYPE of string * term option * ty
      | DYNAMIC of ty * boxed
      | EXN of {exnName: string, hasArg: bool}
      | EXNTAG
      | FUN of {closure: boxed, ty: ty}
      | IENVMAP of (int * term) list
      | INT32 of int
      | INT16 of int16
      | INT64 of int64
      | INT8 of int8
      | INTERNAL
      | INTINF of intInf
      | LIST of term list
      | NULL
      | NULL_WITHTy of ty
      | OPAQUE
      | OPTION of term option * ty
      | PTR of word64
      | REAL64 of real
      | REAL32 of real32
      | RECORDLABEL of RecordLabel.label
      | RECORDLABELMAP of (RecordLabel.label * term) list
      | RECORD of term RecordLabel.Map.map
      | REF of ty * boxed
      | REF_PRINT of term
      | SENVMAP of (string * term) list
      | STRING of string
      | VOID
      | VOID_WITHTy of ty
      | UNIT
      | UNPRINTABLE
```

```
    | VECTOR of ty * boxed
    | VECTOR_PRINT of term vector
    | WORD32 of word
    | WORD16 of word16
    | WORD64 of word64
    | WORD8 of word8
  datatype ty =
      ARRAYty of ty
    | BOOLty
    | BOTTOMty
    | BOXEDty
    | BOUNDVARty of BoundTypeVarID.id
    | CHARty
    | CODEPTRty
    | CONSTRUCTty
        of {args: ty list,
            conSet: ty option SEnv.map,
            id: ReifiedTy.typId,
            layout: ReifiedTy.layout,
            longsymbol: {loc: Loc.pos * Loc.pos, string: string} list,
            size: int}
    | DATATYPEty
        of {args: ty list,
            id: ReifiedTy.typId,
            layout: ReifiedTy.layout,
            longsymbol: {loc: Loc.pos * Loc.pos, string: string} list,
            size: int}
    | DUMMYty of {boxed: bool, size: word}
    | DYNAMICty of ty
    | ERRORty
    | EXNTAGty
    | EXNty
    | FUNMty of ty list * ty
    | IENVMAPty of ty
    | INT16ty
    | INT64ty
    | INT8ty
    | INTERNALty
    | INTINFty
    | INT32ty
    | LISTty of ty
    | OPAQUEty
        of {args: ty list,
            id: ReifiedTy.typId,
            longsymbol: {loc: Loc.pos * Loc.pos, string: string} list,
            size: int}
    | OPTIONty of ty
    | POLYty
        of {body: ty,
            boundenv: BoundTypeVarID.id BoundTypeVarID.Map.map}
    | PTRty of ty
    | REAL32ty
    | REAL64ty
    | RECORDLABELty
    | RECORDLABELMAPty of ty
    | RECORDty of ty RecordLabel.Map.map
    | REFty of ty
    | SENVMAPty of ty
```

```
    | STRINGty
    | TYVARty
    | UNITty
    | VECTORty of ty
    | VOIDty
    | WORD16ty
    | WORD64ty
    | WORD8ty
    | WORD32ty
  type 'a dyn (= boxed)
  type void (= unit)
  type dynamic = void dyn
  exception RuntimeTypeError
  val dynamic = fn : ['a#reify. 'a -> void dyn]
  val dynamicToString = fn : void dyn -> string
  val dynamicToTerm = fn : void dyn -> term
  val dynamicToTy = fn : void dyn -> ty
  val dynamicToTyString = fn : void dyn -> string
  val format = fn : ['a#reify. 'a -> string]
  val fromJson = fn : string -> void dyn
  val fromJsonFile = fn : string -> void dyn
  val join = fn : void dyn * void dyn -> void dyn
  val pp = fn : ['a#reify. 'a -> unit]
  val termToDynamic = fn : term -> void dyn
  val termToString = fn : term -> string
  val termToTy = fn : term -> ty
  val toJson = fn : ['a. 'a dyn -> string]
  val tyToString = fn : ty -> string
  val valueToJson = fn : ['a#reify. 'a -> string]
  val view = fn : ['a#reify. 'a dyn -> 'a]
  ...
end
```

Chapter 28

The `smlsharp` command

The `smlsharp` command invokes the SML# compiler to translate SML# programs into machine code, generate object files, and link them together into an executable file. Similarly to the traditional C compiler driver command, the user can do only one step of this compilation sequence by specifying a mode switch to the `smlsharp` command. The interactive session is also invoked by the `smlsharp` command.

The synopsis of `smlsharp` command is as follows:

```
smlsharp [ option ··· ] [ -- ] [ inputFile ··· ]
```

Each argument of `smlsharp` command is either a command line option or input file name. A command line usually starts with a minus sign (`-`). The order of command line options does not matter except for a few options such as `-I` and `-L`. Optional `--` indicates the end of the command line option sequence; any arguments after `--` are not interpreted as options. Arguments other than options are input file names. The input file names may be interleaved with the option sequence. Regardless of the order of options and input file names, all command line options are interpreted at first, and then the effect of the options are applied to all input files.

In addition, several environment variables affects the behavior of the `smlsharp` command. Such environment variables also affects not only `smlsharp` but also all programs compiled by the SML# compiler.

In this chapter, we introduce `smlsharp`'s command line options and environment variables separately for each category.

28.1 Mode switch

The following options specifies an execution mode of the `smlsharp` command. At most one of these options may be specified in a command line.

`--help` Print the help message and exit.

`-fsyntax-only` Check the syntax of the given `sml` files and `smi` files and exit. Specifying this option together with `-o` option is not allowed. The result of the syntax check is reported by error messages and exit status.

`-ftypecheck-only` Perform the typecheck of the given `sml` files and `smi` files and exit. Specifying this option together with `-o` option is not allowed. The result of the syntax check is reported by error messages and exit status.

`-S` Compile the given `sml` files and generate assembly code files. By default, the name of the output file is obtained by replacing the `.sml` suffix of the input file name with `.s`. If only one input file is given, you may specify the name of output file by `-o` option.

`-c` Compile the given `sml` files and generate object files. By default, the name of the output file is obtained by replacing the `.sml` suffix of the input file name with `.o`. If only one input file is given, you may specify the name of output file by `-o` option.

`-Mm` Generate a Makefile that compiles and links programs, the entries of which are specified by the given `.smi` files as input files. It computes all file dependencies for compiling and linking from the given `.smi` files. If your project does not use source-generating tools such as ml-lex and ml-yacc, this mode generates a complete Makefile for that project. If `-o` is specified together, the result is written in the specified output file instead of the standard output.

`-MMm` Same as `-Mm` except that files in the standard library are omitted.

`-M` Print the list of source files required to compile each of the given `sml` files in the format of Makefile rule. If `-o` is specified together, the result is written in the specified output file instead of the standard output.

`-MM` Same as `-M` except that files in the standard library are omitted.

`-Ml` Print the list of object files required when linking a program with each of the given `smi` files. If `-o` is specified together, the result is written in the specified output file instead of the standard output.

`-MMl` Same as `-Ml` except that files in the standard library are omitted.

If none of the above options is specified, the execution mode of `smlsharp` is decided by the input files.

- If no input file is given, it starts the interactive session.
- Otherwise, `smlsharp` goes into link mode. In this mode, the input file list may include at most one `.smi` or `.sml` file. If a `.smi` file is given, it computes the list of `.smi` files by tracing `_require` relationship from the given `.smi` file, searches for an object file corresponding to each `.smi` file in the list, and links the object files found all together. For details of the object file search, see the description of `-filemap` option. If a `.sml` file is given, it compiles the given `.sml` file to an object file and then obtains the list of object files as if its corresponding `.smi` file is given as a input file. Any other input files must be object or library files that the system linker accepts. The `smlsharp` command invokes the system linker and passes the list of input files to the linker.

  Note that in link mode, similarly to the system linker, the order of input files is significant. An object file that has unresolved symbols must precede those that provide them.

  The file name of the executable program is `a.out` by default. You may specify it by `-o` option.

28.2 Common options for all modes

`-o` *filename* Output the result to the file named *filename*. The contents of the output file varies from modes.

`-v` Be verbose. If this option is set, all command lines of external commands invoked by `smlsharp` command and their output are printed. If this option is set without any input file, it prints the SML# version and exits.

28.3 Compile options

The following options controls file search and code generation of the SML# compiler. Options in this category also affects the interactive session and link mode.

`-I`*dir* Add the directory *dir* to the search path of `.smi` files. If multiple `-I` options are specified, the SML# compiler searches `.smi` file in the given order. Note that this option is also effective in link mode for computing object file list to be linked.

`-nostdpath` Search for `.smi` files only in directories specified by `-I` options.

`-O`, `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Oz` Change the optimization level. `-O0` disables optimization. `-O1` to `-O3` enable optimization. Bigger number allows more aggressive optimization. `-O` is an alias of `-O2`. `-Os` and `-Oz` makes code size smaller. If more than one of these options are specified, all of them except for the last one are ignored.

`--target=`*target*, `-mcmodel` *model*, `-march` *arch*, `-mcpu` *cpu*, `-mattr` *attrs* Set the target, code model, architecture, CPU, and code generation attributes of LLVM's code generator. *model* must be either `small`, `medium`, `large`, or `kernel`. *attrs* is a comma-separated list of attributes. See LLVM manual or help of `llc` command for details.

`-fpic`, `-fPIC`, `-fno-pic`, `-mdynamic-no-pic` Set the code relocation model. `-fpic` is an alias of `-fPIC`. `-fPIC` forces the code generator to generate relocatable code. `-fno-pic` is for non-relocatable code. `-mdynamic-no-pic` means non-relocatable code containing position independent external symbols. See LLVM manual for details. If none of them is specified, the default model is selected depending on the target platform.

`-Xllc` *arg*, `-Xopt` *arg* Add *arg* to the additional arguments to be passed to LLVM's `llc` and `opt` command, which are invoked by `smlsharp` for code generation. If more than one of them are given, all of them are passed to the LLVM commands in the given order.

`-emit-llvm` Output LLVM IR instead of native code when creating a file of compilation result. If in `-S` mode, `smlsharp` generates text LLVM IR code instead of assembly code. The default suffix of the output file is changed from `.s` to `.ll`. If in `-c` mode, it generates LLVM bitcode file instead of object file. The default suffix of the output file is changed from `.o` to `.bc`.

28.4 Link options

The following options control the behavior of linker. All options in this category also affects the interactive session.

`-l`*library* Link with the specified library. In interactive session, this option allows the user to import functions in the given library interactively.

`-L`*dir* Add the directory *dir* to the library search path. If more than two `-L` options are specified, it searches for libraries in the given order.

`-nostdlib` Search for libraries only in directories specified by `-L` options.

`-c++` Use C++ compiler driver instead of C compiler driver for linking. This is needed to link a SML# program with C++ libraries.

`-Wl` *args*, `-Xlinker` *arg* Add *arg*s to the additional arguments to be passed to the C/C++ compiler driver command when linking. *args* of `-Wl` is a comma-separated list of arguments. `-Xlinker` specifies just one argument. If more than two of them are specified, all of them are passed to C/C++ compiler driver command in the given order.

`-filemap` *filename* Use *filename* as the map of filenames that the compiler uses for searching for files. The `smlsharp` command attempts to obtain object file name by replacing the `.smi` suffix of a `.smi` file with `.o`. By specifying this option, you can change this correspondence. Each line of the given file must be a `=` sign followed by a filename that the compiler generates followed by its corresponding file, separated by white-spaces. The map file must contain all files that would be visited by the compiler. If a file not in the map file is needed, it causes an error.

28.5 Interactive mode options

`-r` *smifile* Load additional `.smi` file for the initial environment. This option is useful if you want to make your own library available in the interactive session. The library indicated by the given `.smi` file must be compiled, in other words, all of the object files constituting the library must be prepared. The `smlsharp` command obtains the object files as if the `.smi` file is specified in link mode, link them to a shared library, and load it through the system facility of dynamic loading.

28.6 Developers' options

The following options are for SML# compiler development.

`-d` [*switch*] Set compiler developers' switch *switch*. if *switch* is omitted, it goes into verbose mode for compiler developers. If `-d` without *switch* is specified together with `--help`, it prints the list of compiler developers' switches.

`-B` *dir* Set the directory containing compiler configuration file (config.mk) and the SML# standard libraries.

28.7 Environment variables

The SML# compiler refers to the following environment variables:

`SMLSHARP_HEAPSIZE` This gives the compiler the hint of the minimum and maximum heap size. The content of this variable must be the minimum size followed by a comma followed by the maximum size. The heap size must be a decimal integer with an optional suffix `K` (killo), `M` (mega), `G` (giga), or `T` (tera). The maximum size may be omitted. If omitted, dynamic heap expansion is disabled. The default setting is "`32M:256M`." If the compiler is aborted due to memory exhaustion, increase the heap size by this environment variable.

`SMLSHARP_VERBOSE` This indicates the verbose level of the SML# runtime in an integer from 0 to 5. Its default is 2. This is provided for SML# compiler developers.

`SMLSHARP_LIBMYSQLCLIENT`, `SMLSHARP_LIBODBC`, `SMLSHARP_LIBPQ` These specify the database libraries that the SML# compiler uses in the interactive mode. Set them if you meet an library load error when using SQL features of SML#.

28.8 Typical examples

28.8.1 Start an interactive session

A typical way to start an interactive session is to execute `smlsharp` command with no argument.

```
$ smlsharp
```

When you use a C library in the interactive session, specify `-l` option with the name of the library. For example, if you want to use zlib library, do

```
$ smlsharp -lz
```

The libraries SML# uses, such as the standard C library and POSIX thread library, are available by default. You can use them without `-l` option.

28.8.2 Compile a program

If your program consists of only one `.sml` file (and its corresponding `.smi` file), just give the `.sml` file to `smlsharp` command and you will obtain its executable file.

```
$ smlsharp foo.sml
```

If you don't like the default name `a.out`, change it by `-o` option.

```
$ smlsharp -o foo foo.sml
```

28.8.3 Compile separately and link a program

A typical way to compile and link a program consisting of multiple files is the following:

```
$ smlsharp -c -O2 foo.sml
$ smlsharp -c -O2 bar.sml
$ smlsharp -c -O2 baz.sml
$ smlsharp foo.smi
```

First, compile each `.sml` file in the project separately in random order by the `smlsharp` command with `-c`. If the compilation of `foo.sml` is done successfully, its object file `foo.o` is generated. After compiling all `.sml` files, link their object files into an executable file by giving the `smlsharp` command the `.smi` file whose top-level code is the entry of the program (in this example, `foo.smi`). At this time, the `smlsharp` command traces all `.smi` files reachable from `foo.smi` through `_require` relationship recursively and obtains the list of object files to be linked by replacing their `.smi` suffix with `.o`.

If you change this default correspondence from `.smi` files to `.o` files, create a map file and give it by `-filemap` option like the following:

```
$ smlsharp -c -O2 -o obj/foo.o foo.sml
$ smlsharp -c -O2 -o obj/bar.o bar.sml
$ smlsharp -c -O2 -o obj/baz.o baz.sml
$ smlsharp -filemap=objmap foo.smi
```

The contents of `objmap` file is given below:

```
= foo.o obj/foo.o
= bar.o obj/bar.o
= baz.o obj/baz.o
```

SML# programs may be linked with arbitrary C/C++ programs and libraries. To link your program with C/C++ libraries, give them to the `smlsharp` command with a `.smi` file. For example, to link `foo.sml` with a C program `util.c` and C libraries `libgl` and `libglu`, do

```
$ smlsharp -c -O2 foo.sml
$ cc -c -O2 util.c
$ smlsharp foo.smi util.o -lgl -lglu
```

28.8.4 Generate a Makefile

it is useful to combine SML# with "make" command so that your program is separately compiled and linked automatically. To use make command, you need to create a Makefile that includes dependencies between source files. By `smlsharp -Mm` command, you can automatically obtain a complete Makefile for building a SML# program automatically.

For example, suppose a project consisting of three `.smi` files `foo.smi`, `bar.smi`, and `baz.smi` (and their corresponding `.sml` files). The `_require` relationship between them is as follows:

- `foo.smi` _requires `bar.smi`.
- `bar.smi` _requires `baz.smi`.

In this case,

```
$ smlsharp -Mm foo.smi -o Makefile
```

generates the following Makefile:

```
SMLSHARP = smlsharp
SMLFLAGS = -O2
LIBS =
all: foo
foo: foo.o bar.o baz.o foo.smi
        $(SMLSHARP) $(LDFLAGS) -o $@ foo.smi $(LIBS)
foo.o: foo.sml foo.smi bar.smi baz.smi
        $(SMLSHARP) $(SMLFLAGS) -o $@ -c u.sml
```

```
bar.o: bar.sml bar.smi baz.smi
        $(SMLSHARP) $(SMLFLAGS) -o $@ -c u.sml
baz.o: baz.sml baz.smi
        $(SMLSHARP) $(SMLFLAGS) -o $@ -c u.sml
```

Then, just do `make` and build your program.

```
  % make
  smlsharp  -o foo.o -c foo.sml
  smlsharp  -o bar.o -c bar.sml
  smlsharp  -o baz.o -c baz.sml
  smlsharp  -o foo foo.smi
```

Every time you modify the program and its `_require` relationship is changed, you need to do `smlsharp -MMm` to update the Makefile.

If your project contains source files of ml-lex and ml-yacc, `smlsharp -Mm` is not enough. You need additional rules that invokes those tools to generate `.sml` files. For example, if your project has a parser `parser.grm`, create Makefile by `smlsharp -Mm` and then add the following rule to the Makefile by hand:

```
parser.grm.sml parser.grm.sig: parser.grm
        ml-yacc parser.grm
parser.grm.sig: parser.grm.sml
```

It should be useful if you separate two Makefiles, one of which is automatically generated by `smlsharp -Mm` and another of which includes the additional rules. A simple way to manage two Makefiles is to organize Makefile with additional rules and `include` of the automatically generated one. For example, create a Makefile like the following:

```
# The default target is "all".
all:

# File dependency is automatically generated and included.
depend.mk:
        smlsharp -Mm foo.smi -o depend.mk
include depend.mk

# Additional rules.
parser.grm.sml parser.grm.sig: parser.grm
        ml-yacc parser.grm
parser.grm.sig: parser.grm.sml
```

Of course, you can exploit your favorite Makefile techniques.

Chapter 29

SML# Run-time data management

SML# adopts the most natural data representation (the data representation that the ABI (Application Binary Interface) of the target platform adpots) for all the basic data types. For example, in x86_64 platform, `int` is a 32-bit integer, and `real` is a 64-bit IEEE754 floating point number. The evaluation of expressions of these types are performed by using appropriate registers. In arrays, tuples, and records, these data are aligned appropriately. The natural data representation is kept in the entire SML# program including polymorphic functions.

SML#'s separate compilation and interoperability with C is realized on top of this natural data representation. Therefore, power users that exploits these features need detailed knowledge about runtime data representation. This chapter describes the runtime data representation and memory management of SML#.

29.1 Runtime representation

Any type constructor that the SML# compiler manages has its "size" on memory, "tag" indicating whether or not GC traces, and "representation" indicating the range and semantics of bits. We refer to the triple of these as the *runtime representation* of a type constructor. The runtime representation of a type is determined by its outermost type constructor. The following are runtime representations of built-in types.

built-in type constructor	size (in bytes)	tag	representation
`int`, `int32`	4	0	signed integer
`int8`	1	0	signed integer
`int16`	2	0	signed integer
`int64`	8	0	signed integer
`word`, `word32`	4	0	unsigned integer
`word8`	1	0	unsigned integer
`word16`	2	0	unsigned integer
`word64`	8	0	unsigned integer
`char`	1	0	unspecified
`real`, `real64`	8	0	IEEE754 floating point number
`real32`	4	0	IEEE754 floating point number
`ptr`	4 or 8	0	`void*` of C
`codeptr`	4 or 8	0	function pointer of C
`unit`	4	0	unique variant
`contag`	4	0	unspecified
`boxed`	4 or 8	1	unspecified
`array`	4 or 8	1	non-null pointer
`vector`	4 or 8	1	non-null pointer
`ref`	4 or 8	1	non-null pointer
`string`	4 or 8	1	non-null pointer
`exn`	4 or 8	1	non-null pointer
record type, tuple type	4 or 8	1	non-null pointer
function type	4 or 8	1	non-null pointer

The size of the type whose size is "4 or 8" depends on whether the target platform is 32-bit or 64-bit.

For any type constructors, their alignments are equal to their size. The alignments should be equal to C's alignments, but current SML# compiler gives alignments constantly regardless of target platform information (this restriction will be refined in the future version of SML#).

The runtime representation of user-defined types defined by `datatype` declaration is calculated from the set of data constructors as follows:

- If the type has only one data constructor without argument, the runtime representation is equal to the `unit` type.
- If the type has two or more data constructor, each of which has no argument, the runtime representation is euqal to the `contag` type.
- Otherwise, the runtime representation is equal to the `boxed` type.

When computing runtime represenatation, the alias type defined by `type` declaration are all expanded. The runtime representation of opaque types introduced by opaque signature constraints is equal to that of the implementation type of the opaque type. The runtime representation of the types declared as opaque types by an interface file is equal to that of the implementation type given by its opaque type declaration in the interface file.

29.2 Effect of garbage collection

Data of type whose tag is 1 are managed under SML#'s memory management system. In what follows, we refer to those data as boxed data. The memory area of boxed data is allocated implicitly when its data constructor is evaluated, and released automatically by SML#'s garbage collector. To pass boxed data to C functions, you need to pay attention to not only their contents but also the timing when the memory is released. In perspective of C functions, boxed data have the following properties:

1. Boxed data are not moved similarly to the data allocated by C's `malloc` function. Therefore, the identity of SML#'s arrays and refs is preserved even in C functions. Since tuple and record values are neigher moved as long as the values are alive.
2. Arguments passed from SML# to C functions are passed without any modification. Neither data conversion, allocation, nor regeneration may occur. Therefore, C function may update SML#'s arrays and refs and SML# program see the effect of C's memory update.
3. Boxed data passed to C function are never released during the C function ends. When C function ends and SML# program has control, the boxed data passed to C is not alive in the SML# program, the boxed data is released.
4. Callback functions are never released until the program ends. It is allowed for C functions to store callback functions to global variables. The value of free variables of callback functions are preserved even if the thread is different between the caller and callback. Note that you need to pay attetion to memory leak when you use callback functions. The SML# compiler guarantees that no memory leak occur if you only use top-level functions as callback functions.
5. SML#'s garbage collector is accurate garbage collector that only scans SML#'s heap and stack. GC may release boxed data to which SML# program cannot refer even if they are stored in C heap and hence C functions can refer to them. Therefore, it is not allowed for C functions to store boxed data to C heap. Since boxed data are alive until C function ends, it is safe to store them in registers and stack frames.

29.3 Effect of unwind jumps

SML# and C/C++ provides jumps that unwinds call stack. In SML# and C++, exceptions are provided. In C, `setjmp` and `longjmp` are provided. These unwind jump mechanism can be used in combination with SML#'s callback functions. Unwind jumps has the following properties:

1. SML#'s exception handling mechanism is implemented through Itanium ABI, similarly to C++'s exception mechanism. Therefore, cleanup handlers may be invoked by SML#'s exceptions. For example, destructors of C++ local variables are executed appropriately when callback SML# function raises an exception.

2. Exceptions raised by SML#'s `raise` expression can be handled only by SML#'s `handle` expression. The `handle` expression only handles SML# exceptions and hence it is not able to handle other kind of unwind jumps.

3. If SML#'s exception is not caught by any `handle` expression, the program aborts.

4. C++ and SML# exceptions works independently of each other with SML# callback functions.

5. C's `setjmp` and `longjmp` also work with SML# callback functions. It is not valid to create any loop in a SML# program by using `setjmp` and `longjmp`.

29.4 Effect of multithreading

SML# allows the users to import C functions that spawns new threads. Even if SML# callback functions are called from a thread that C function creates, the SML# runtime detects the new threads and runs SML# programs concurrently in multiple threads. Similarly to the C programs, SML# uses the multithread support provided by the underlying operating system. If operating system supports parallel execution of multiple threads on multicore CPUs, SML# program using multiple threads runs in parallel.

Threads of SML# program shares a unique SML# heap. Interthread communication can be realized by updating arrays. Exclusive execution is performed by calling C function that realizes exclusive execution. Since SML# never move data on heap, you are allowed to allocate C's semaphores and mutexes on SML#'s heap.

This is all abort SML#'s multithread support. Current SML# does not provide high-level thread library, but it is open to the users to write it in SML#. The future version of SML# will be likely to provide a parallel programming framework.

Part IV

Programming Tools

Chapter 30

A parser generator smlyacc and smllex

A paper generator `smlyacc` and a lexer generator `smllex` are bundled with the SML# distribution. The original programs were developed by Andrew W. Appel and David R. Tarditi Jr., and distributed for the SML/NJ system. The license and documents are found in `src/ml-yacc/COPYRIGHT` and `src/ml-yacc/doc/mlyacc.pdf`. They are ported to the SML# separate compile system. The readers are referred to the `src/ml-yacc/doc/mlyacc.pdf` for for source file syntax, including `smlyacc` grammar rules and `smllex` regular expressions.

This document describes how to run the programs and and to use the generated parser and lexer in SML#.

30.1 The generated files

The SML# installer will compile and install `smlyacc` and `smllex` commands, which are used to generate the SML# system. These commands generates the following SML# source files.

command	input file	generated files	Contents
`smlyacc`	⟨*YaccInputFileName*⟩ `.grm`	⟨*YaccInputFileName*⟩ `.grm.sml`	A Parser Program
		⟨*YaccInputFileName*⟩ `.grm.sig`	Parser's Token signature
		⟨*YaccInputFileName*⟩ `.grm.desc`	LR automaton state descriptions
`smllex`	⟨*LexInputFileName*⟩ `.lex`	⟨*LexInputFileName*⟩ `.lex.sml`	A Lexical analyzer

- `smlyacc` command only takes an input file name ⟨*YaccInputFileName*⟩ .grm. The output file name can be specified with the following environment variable:

  `SMLYACC_OUTPUT=`⟨*YaccOutputFileName*⟩ *.sml*

  The specification of the output file includes its suffix (`.sml`). With this environment variable is set, `smlyacc` generates the token signature at the top of the output file.

- `smllex` command only takes an input file name ⟨*LexInputFileName*⟩ .lex. The output file name can be specified with the following environment variable:

  `SMLLEX_OUTPUT=`⟨*LexOutputFileName*⟩ *.sml*

  The specification of the output file includes its suffix (`.sml`).

30.2 The structure of a smlyacc input file

An input file of `smlyacc` has the following general structure.

⟨*SML# code for user declarations* ⟩
%%
⟨*YACC declarations for grammar rule interpretation*⟩
%%
⟨*descriptions of grammar rules and their attributes*⟩

1. The SML# code for user declarations section specifies any user level code which will be used in attribute specifications in the rule section.

2. The YACC declarations for grammar rule interpretation section include meta-level declarations for grammar rules such as non-terminal associations, and directives for `smlyacc`. The meta-level declarations for grammar rules are the same as the standard YACC system. The reader is referred to `src/ml-yacc/doc/mlyacc.pdf` for the details.

   When using the generated parser source in SML#, the following directives should be specified.

   ```
   %name <Name>
   %header (structure <Name>)
   %eop EOF SEMICOLON
   %pos int
   %term EOF
      | CHAR of char
       ...
   %nonterm id of Symbol.symbol
      | longid of Symbol.longsymbol
       ...
   ```

   - `%name`specification. It specifies the name of the parser. The name `<Name>_TOKENS` will be used is the name of the generated toke structure.
   - `%header`specification. `smlyacc` will generate the parser program as a structure body in the following format.

     ```
     = struct
        ...
     end
     ```

     `%header()` specifies the code fragment that will be In order to use the generated parser in the SML# separate compilation mode, it needs to include the preamble of the structure declaration as above.
   - `%pos`specificaion. It specifies the type of the text position used in `smllex`.
   - `%eop`specificaion. It defines the set of terminal symbols that end the parsing.
   - `%term`specificaion. The set of terminal symbol names is defined as a form of datatype constructor. For each terminal symbol name, `smlyacc` generates a token forming function of the forms `EOF : pos * pos -> token` (without argument) or `CHAR : char * pos * pos -> token` (with arguments). The generated token functions are put into the `<Name>_TOKENS` structure and is shared by `smllex`.
   - `%nonterm`specificaion. The set of non-terminal symbols are defined as a form of constructor having the type of its attribute. The attribute type is the type of expression associated to the grammar rule of that non terminal symbol.

3. The descriptions of grammar rules and their attributes section defines the sets of terminal and non-terminal symbols, and the set of grammar rules and their associated action rules. The syntax for the rules as the same as the standard YACC system. The reader is referred to `src/ml-yacc/doc/mlyacc.pdf` for the details.

30.3 The structure of a smlyacc output file and the interface file specification

`smlyacc` generates a file ⟨*YaccInputFile*⟩`.grm.sml` containing one structure of the following signature.

```
signature ML_LRVALS =
  sig
     structure Tokens : ML_TOKENS
     structure Parser : PARSER
     sharing type Parser.token = Tokens.token
  end
```

This signature and other supporting library for `smlyacc` are collected in `smlyacc-lib.smi`. In order to use the generated parser program file, the following interface file must be written.

```
_require "basis.smi"
_require "ml-yacc-lib.smi"

structure <Name> =
struct
  structure Parser =
  struct
    type token (= boxed)
    type stream (= boxed)
    type result = Absyn.parseresult
    type pos = int
    type arg = unit
    exception ParseError
    val makeStream : {lexer:unit -> token} -> stream
    val consStream : token * stream -> stream
    val getStream : stream -> token * stream
    val sameToken : token * token -> bool
    val parse : {lookahead:int,
                 stream:stream,
                 error: (string * pos * pos -> unit),
                 arg: arg}
                -> result * stream
  end
  structure Tokens =
  struct
    type pos = Parser.pos
    type token = Parser.token
    <the set of Token forming functions>
     ...
    val EOF: word *  pos * pos -> token
    val CHAR: string *  pos * pos -> token
    ....
  end
end
```

In `Parser` structure, `pos`, `arg`, `result` types are those that are specified in ⟨*YaccInputFileName*⟩ `.grm`. The other components are fixed and should be specified as above. To write the `Token` structure, copy the contents of the ⟨*YaccInputFileName*⟩ `.grm.sig` signature generated by `smlyacc`.

The components are described below.

- `token` type. The abstract type for the token returned by the lexer generated by `smllex`.
- `stream` type. The abstract type for the input stream of the parser.
- `result` type. The type of the output of the parser. It should be the same as the type of the attribute of the top-level grammar rule specified in the ⟨*YaccInputFile*⟩ `.grm` file. の属の性の型と同一である.
- `pos` type. The position data type used by the lexer.
- `arg` type. The additional argument type for the parser specified in the ⟨*YaccInputFile*⟩ `.grm` file.
- `ParseError` exception. Parse error exception generated by the parser.
- `makeStream` function. It takes a lexer and return the input stream for the parser.
- `consStream` function. This up-push one token to the given input stream.
- `getStream` function. It returns the first token in the current input stream.

- `sameToken` function. Token equality checking function.
- `parse` function. The parser function. It takes the current input stream and returns the generated abstract syntax tree and the rest of the input stream.

30.4 The structure of a smllex input file

An input file of `smllex` has the following general structure.

⟨*user declarations*⟩
%%
⟨*LEX declarations*⟩
%%
⟨*descriptions of regular expressions and their attributes*⟩

1. The user declaration section The SML# code for user declarations section specifies the types that are used in lexical analysis and any other user level code which will be used in attribute specifications in the regular expression section.

   The mandatory specifications are the following.

   ```
   type lexresult = ...
   fun eof () = ...
   ```

   - `lexresult` type. The type of the value of returned by the lexer when the lexer accepts a regular expression. When using with the `smlyacc`, this is usually the token type defined in ⟨*YaccInputFile*⟩ `.grm` file.
   - `eof` function. This is the function that is called when the lexical analyzer detects the end of file. It usually returns the value (of type `lexresult`) that represent the end of file token.

2. The LEX declaration section

   This section specifies directives for `smllex` functions and auxiliary definitions for specifying regular expressions in the regular expression section. The syntax of auxiliary definitions for regular expressions are the same as the standard LEX system.

   `smllex` directives include the following.

   - `%structure` 宣言. `smllex` generate a lexical analyzer as a structure. This declaration specifies the name of the structure as follows.

     ```
     %structure MLLex
     ```

   - `%arg` 宣言. This specifies an extra argument to be passed to the lexical analyzer. It can be omitted if no extra argument is needed.
   - `%full` 宣言. This specifies that the generated lexer handles 8 bit character.

3. The regular expression definition section This section defines the set of regular expressions to be accepted.

For the details of the specification, consult the document `src/ml-lex/doc/mllex.pdf` in the SML# source code distribution.

30.5 The interface file for the generated lexer

`smllex` generate a structure containing a lexer creating function `makeLexer`. In order to use the generated lexer program file, a interface file must be written. The minimal interface is of the following form.

```
_require "basis.smi"
_require "coreML.grm.smi"

structure CoreMLLex =
struct
  val makeLexer : (int -> string) -> unit -> CoreML.Tokens.token
end
```

The first argument to the `makeLexer` is a function that takes a number and return an input string of that specified size. Applying this function to an input function like `makeLexer (fn n => TextIO.inputN(TextIO.stdIn,n)` generates a lexer function of type `unit -> token`. This lexer function can be specified as an argument to the `makeStream` function of the parser to obtain a token `stream` for the parser input.

User declarations such as extra arguments are placed in the `UserDeclarations` structure. Its interface file should be of the following form.

```
_require "basis.smi"
_require "<YaccInputFile>.grm.smi"

structure MLLex =
struct
  structure UserDeclarations =
  struct
    type token = <YaccName>.Tokens.token
    type pos = <YaccName>.Tokens.pos
    type arg (= boxed)
  end
  val makeLexer
      : (int -> string) -> UserDeclarations.arg -> unit -> UserDeclarations.token
end
```

Part V

SML# Internals and Data Structures

Chapter 31

Preface

This part presents internals and data structures of the SML# compiler. The objective of this part is provide the readers who have appropriate backgrounds on functional programming languages to understand the details of the SML# compiler.

Although the main intended audience are compiler developers and researchers, who are interested in extending SML# or developing new compilers, efforts have been made to make this document should a reference document for general readers who are interested in high level programming language compilers.

There are excellent codes in open-source software, many of which we truly respect. One of the problems we often feels against open-source culture is the lack of effort and enthusiasm to document the structures and functions of these excellent codes. The contention that code written in a declarative language is itself self-documenting is a vain argument for large and complicated systems containing hundreds of thousands of lines. A large and complicated software system inevitably contains implicit assumptions and various meta-level data that control behavior of multiple components of the system. To understand the system, it is therefore necessary to understand the encoded semantics of those control data and the flow of control of the multiple related components. Currently, the only way to solve this problem is to document the system. We further believe that a detailed document of a large software system could itself be a valuable asset for open-source community.

In this part, we attempt to provide such a document on the SML# system, taking "VAX/VMS internals and data structures" [4] as a model. In addition to descriptions of the code, we try to provide the design rationale, relevant theory, algorithm, and implementation techniques underlying the code.

Part V contains the following.

1. Chapter 32 describes the SML# distribution package.
2. Chapter 33 outline the control flow of the SML# compiler.
3. Each of the following chapters describes the structure and function of each of the modules in the order of compilation flow described in Chapter 33.
4.
5. we shall continue writing the entire internals, and shall uploaded them as they become ready ⋯
6.

Chapter 32

The SML# Source Distribution Package

SML# distribution package `smlsharp-4.2.0.tar.gz` contains the sources and other resources for the SML# compiler, the Standard ML Basis library, and several supporting tools.

32.1 The structure of the source package

SML# source distribution package `smlsharp-4.2.0.tar.gz` consists of the directories and files.

- Directories

`benchmark/`	benchmark programs
`precompiled/`	assembry source archives for `minismlsharp`
`sample/`	sample programs
`src/`	the SML# source files
`test/`	test programs and data

- files

`Changes`	the release notes
`INSTALL`	install instructions
`LICENSE`	the SML# license
`Makefile.in`	`Makefile` template
`RELEASE_DATE`	the release date
`VERSION`	the current version
`config.h.in`	`config.h` template.
`config.mk.in`	`config.mk` template containing configuration parameters used in `Makefile`
`configure.ac`	input to `autoconf`
`depend.mk`	source file dependency
`files.mk`	file sets definition
`mkdepend`	A shell script to generate `depend.mk`
`precompile.mk`	a `make` script for re-generating `precompiled/` archives

32.2 The SML# Source Tree

The SML# source directory `src` consists of the following directories and files.

- Directories
  - SML# compiler source

`basis/`	the Standard ML Basis Library
`compiler/`	the SML# compiler
`config/`	the library to access system parameter set by `configure`
`ffi/`	the support library for direct C interface
`llvm/`	the LLVM code generation library
`runtime/`	the runtime system
`sql/`	the SQL integration support library
`thread/`	the thread support library
`unix-utils/`	the library for UNIX basic commands

  - SML# tools

`smlformat/`	`smlformat`the pretty printer generator
`smlunit/`	the unit test library

    These are general purpose tools we have developed for SML#. Since they are relatively small system independent of the SML# compiler, we omit their description in this current document.

  - third-party codes

`ml-lex/`	a lexer generator
`ml-yacc/`	a parser generator
`smlnj/`	smlnj source files used in `basis/`
`smlnj-lib/`	smlnj utility library

    These third-party codes are ported to SML#. They are are included here with their licenses. `make` system compiles them with SML# source code to build the SML# system. This document does not describe these code.

- Files

`basis.smi`	the interface file of the Standard ML Basis Library
`builtin.smi`	the interface file to bind builtin data
`config.mk`	definition of environment variables of the compiler command
`config.mk.in`	the template file of `config.mk`
`config.sed`	the sed script to generate `config.mk`
`ffi.smi`	the interface file of the direct C interface
`foreach.smi`	the interface file of data parallel execution
`json.smi`	the interface file of JSON support
`ml-yacc-lib.smi`	the interface file of the smlyacc library
`prelude.smi`	the interface file of interactive mode
`reifyTy.smi`	the interface file for reflection
`smlformat-lib.smi`	the interface file of the smlformat library
`smlnj-lib.smi`	the interface file of `smlnj-lib`
`smlunit-lib.smi`	the interface file of smlunit
`sql.smi`	the interface file of the SQL integration support
`thread.smi`	the interface file of the thread support

32.3 `compiler` directory

The `compiler` directory is divided into sub-directory, as shown.

- Directories

  - `compilePhases/` : compilation phases

bitmapcompilation/	explicit layout-bitmap generation
cconvcompile/	type-directed calling-convention generation
closureconversion/	closure conversion
datatypecompilation/	datalayout computation
elaborate/	syntactic elaboration
fficompilation/	C interface generation for higher-order functions
llvmemit/	llvm code emission
llvmgen/	
loadfile/	
machinecodegen/	law-level code generation
main/	
matchcompilation/	pattern matching compilation
nameevaluation/	name evaluation and module compilation
parser/	parser
recordcalcoptimization/	typed record calculus optimization
recordcompilation/	type-directed record compilation
stackallocation/	stack frame allocation
toplevel/	
typedcalcoptimization/	typed intermediate language optimization
typedelaboration/	
typeinference/	type definition, uncurry optimization
valrecoptimization/	mutual recursive function optimization

– compilerIRs/ : compiler intermediate representations

absyn/	Abstract syntax tree
anormal/	A-normal forms
bitmapcalc/	the language with explicit layout bitmap
closurecalc/	the language with explicit closure allocation
idcalc/	the language without static scoping
llvmir/	LLVM IR
machinecode/	the register transfer language
patterncalc/	the untyped intermediate language
recordcalc/	the polymorphic record calculus
runtimecalc/	the language with runtime types
typedcalc/	the typed intermediate language
typedlambda/	the typed lambda calculus

– data/ : types, constants and other data

builtin/	compiler built-in data
constantterm/	constants
control/	parameters controlling compiler function
name/	runtime code labels
runtimetypes/	runtime types
symbols/	representation of variables, labels etc
types/	type representations

– extensions/ : various compile functions

concurrencysupport/	concurrent thread support
debug/	debugging support
foreach/	massively parallel `_foreach` statement support
format-utils/	formater library for `smlformat`
json/	JSON manipulation support
reflection/	compile-time reflection support
usererror/	compiler error handling
userlevelprimitive/	compiler extension via user code

– libs/ : libraries used by the compiler

`digest/`& SHA hashing library	
`env/`	dictionary utilities
`heapdump/`	heap image dump function
`ids/`	counter utilities
`interactivePrinter/`	interactive printers
`list-utils/`	list utilities
`toolchain/`	UNIX toolchain commands
`util/`	miscellaneous utility functions

- Files

`minismlsharp.smi`	the interface file of `minismlshap`
`minismlsharp.sml`	the top-level of the compiler to compile `smlsharp`
`smlsharp.smi`	the interface file of `smlsharp`
`smlsharp.sml`	`smlsharp` top-level

Each of directories, including `compiler` sub-directories, contains the `main` sub-directory, where the source files are located. For example, the source files for abstract syntax trees are placed in `compilerIRs/absyn/main/`. Source files include program files with `.sml` suffix and the corresponding interface files of the same names with `.smi` suffix.

Files with following suffix are input files of source file generation tools.

`.ppg`	an input file to `smlformat` pretty-printer generator.
`.grm`	an `smlyacc` input file
`.lex`	an `smlyacc` input file

An interface file having of one of those names with `.smi` suffix describes the interface of the generated source program.

32.4 `basis` directory

`basis/` is the Standard ML Basis Library source file directory, whose `main` sub-directory contains the following files.

1. Signature files

```
ARRAY.sig
ARRAY_SLICE.sig
BIN_IO.sig
BOOL.sig
BYTE.sig
CHAR.sig
COMMAND_LINE.sig
DATE.sig
GENERAL.sig
IEEE_REAL.sig
IMPERATIVE_IO.sig
INTEGER.sig
INT_INF.sig
IO.sig
LIST.sig
LIST_PAIR.sig
MATH.sig
MONO_ARRAY.sig
MONO_ARRAY_SLICE.sig
MONO_VECTOR.sig
MONO_VECTOR_SLICE.sig
OPTION.sig
OS.sig
OS_FILE_SYS.sig
OS_IO.sig
OS_PATH.sig
OS_PROCESS.sig
PRIM_IO.sig
REAL.sig
STREAM_IO.sig
STRING.sig
STRING_CVT.sig
SUBSTRING.sig
TEXT.sig
TEXT_IO.sig
TEXT_STREAM_IO.sig
TIME.sig
TIMER.sig
VECTOR.sig
VECTOR_SLICE.sig
WORD.sig
```

2. Common generic codes

```
ArraySlice_common.sml
Array_common.sml
VectorSlice_common.sml
Vector_common.sml
```

3. Structure files directories

   For each of the following names, the directory contains the program file with `.sml` suffix and the interface file with `.smi` suffix.

```
Array
ArraySlice
Bool
Byte
Char
CharArray
CharArraySlice
CharVector
CharVectorSlice
CommandLine
Date
General
IEEEReal
IO
Int
IntInf
List
ListPair
OS
Option
Real
Real32
String
StringCvt
Substring
Text
Time
Timer
Vector
VectorSlice
Word
Word8
Word8Array
Word8ArraySlice
Word8Vector
Word8VectorSlice
```

4. SML# support files

   The following files provide SML# specific low-level support for efficient implementation of the basis library.

`SMLSharp_Runtime`	SML# runtime primitives
`SMLSharp_OSFileSys`	primitives for OS structures
`SMLSharp_OSIO`	primitives for IO structures
`SMLSharp_OSProcess`	primitives for `OS.Process`
`SMLSharp_RealClass`	primitives for `Real` structures
`SMLSharp_ScanChar`	primitives for `scan` functions

5. top-level

`toplevel.sml`	top-level declarations
`toplevel.smi`	the interface file of the top-level declarations

Chapter 33

Control structure of the compiler

This chapter describes the control structure of the SML# compiler.

1. Source code locations
   - `src/compiler/minismlsharp.{smi,sml}` files `src/compiler/smlsharp.{smi,sml}` files
   - `src/compiler/compilePhase/main` directory
   - `src/compiler/compilePhase/toplevel` directory
2. Overview
   (a) Compiler starat up by `src/compiler/minismlsharp.sml`.
   (b) The compiler command main functions by `src/compiler/compilePhase/main` module.
   (c) The top-level compile phase processing by `src/compiler/compilePhase/toplevel`.

33.1 Compiler Start-up

The SML# compiler is an SML# program separately compiled and liked by the SML# compiler. the main function of an SML# program, initially called from the SML# runtime system, sequentially evaluets the top-level declarations of the set of source files in an order that respects the dependency among the source files. The set of source files of an SML# program consists of the source file corresponding to the root interface file specified as a command line argument to the link mode SML# command, and the source files correspondig to the interface files that are (directly and indirectly) referenced from the root interface file.

The SML# compiler command is linked by the following shell command written in `Makefile` file for the target `src/compiler/smlsharp`.

```
        $(SMLSHARP_ENV) $(SMLSHARP_STAGE1) -Bsrc -nostdpath $(SMLFLAGS) \
 -filemap=filemap \
 $(RDYNAMIC_LDFLAGS) $(LLVM_SMLSHARP_LDFLAGS) --link-all \
 $(srcdir)/src/compiler/smlsharp.smi \
 $(COMPILER_SUPPORT_OBJECTS) $(LLVM_LIBS) $(LLVM_SYSLIBS) -o $@
```

In the standard configuration, the shell variable `SMLSHARP_STAGE1` is bound to `minismlsharp` commande, which is the same as `smlsharp` except that it links a minimal set of libraries. The top-lebel source file of the SML# compiler is

```
./src/compiler/smlsharp.sml
```

which corresponds to the root interface file `./src/compiler/smlsharp.smi` specified in the avove shell command. This file consists of the following four lines.

```
val commandLineName = CommandLine.name ()
val commandArgs = CommandLine.arguments ()
val status = Main.main (commandLineName, commandArgs)
val () = OS.Process.exit status
```

`Main.main` is the following `main` function written in the file `Main.sml` in `comiler/compilePhases/main/main` directory.

```
val main : string * string list -> OS.Process.status
```

This is the top-level function of the SML# compiler. This function takes a command name (`smlsharp`) string and command line parameter string list, and perform compilations and linking according to the parameter specification.

33.2 The compiler command main function

The main function of the SML# command is the follwing function in the directory `comiler/compilePhases/main/main/`.

```
val main : string * string list -> OS.Process.status
```

The argument pair consists of the command name (`smlsharp` or `minismlsharp`) and the command line argument kist. This function invokes `command` function to do the following.

1. Parse the arugment list and determine the command mode and the set of command options.
2. Perfoms one of the following depending on the command mode.
   (a) Compile source files.
   (b) Compile and link one soruce file.
   (c) Link a system specified by a root `.smi` file.
   (d) Generate dependency and Makefile for a soure file.
   (e) Print various information.

   Compiling a source file is done by `compileSMLFile` function in the following steps.

   - Open the source file.
   - Generate a compile ontext (`topContext`) prepared by the `command` function.
   - Compile the source file by the `Top.compile` function in `comiler/compilePhases/top/main/`, which has the following type.

     ```
     val compile
         : LLVMUtils.compile_options
           -> options
           -> toplevelContext
           -> Parser.input
           -> InterfaceName.dependency * result
     ```

     The return value type `InterfacceName.dependency` represent the set of interface files used by the source file, and `result` denotes the generated object file.

   Compile and link is done by `link` function in the following steps.

   (a) If the input is a soruce file (`.sml` file), then compile it to obtain an object file and the set of dependent interface files. If the input file is an interface file (`.smi` file), determine the root object file, and obtain the set of dependent interface files by calling `loadSMI` function.
   (b) Link the root object file, the set of dependent object files and the referenced library files by calling the system linker.

33.3 The compiler toplevel

The toplevel function of the SML# compiler is function `Top.compile` definde in `comiler/compilePhases/toplevel/main/To` file. Its has the following type.

```
val compile :
    : LLVMUtils.compile_options
      -> options
      -> toplevelContext
      -> Parser.input
      -> InterfaceName.dependency * result
```

It takes LLVM code generation options, compile options, compile context, input stream of the source file, compiles the source and returns the set of dependent interface files and the result object file. This function is called from `compileSML` function in `Main.sml` file. It compiles the source file by calling the funtions of the compilation phases as shown in the following table.

Compile step	Compile Phase Function	Source Language	Target Language
1	`Parser.parse`	(input stream)	`Absyn.absyn`
2	`LoadFile.load`	`Absyn.absyn`	`AbsynInterface.co`
3	`Elaborator.elaborate`	`AbsynInterface.compile_unit`	`PatternCalcInterf`
4	`NameEval.nameEval`	`PatternCalcInterface.compile_unit`	`IDCalc.topdecl`
5	`TypedElaboration.elaborate`	`IDCalc.topdecl`	`IDCalc.topdecl`
6	`VALREC_Optimizer.optimize`	`IDCalc.topdecl`	`IDCalc.topdecl`
7	`InferTypes.typeinf`	`IDCalc.topdecl`	`TypedCalc.tpdecl`
8	`UncurryFundecl.optimize`	`TypedCalc.tpdecl list`	`TypedCalc.tpdecl`
9	`PolyTyElimination.compile`	`TypedCalc.tpdecl list`	`TypedCalc.tpdecl`
10	`TPOptimize.optimize`	`TypedCalc.tpdecl list`	`TypedCalc.tpdecl`
11	`MatchCompiler.compile`	`TypedCalc.tpdecl list`	`RecordCalc.rcdecl`
12	`FFICompilation.compile`	`RecordCalc.rcdecl list`	`RecordCalc.rcdecl`
13	`RecordCompilation.compile`	`RecordCalc.rcdecl list`	`RecordCalc.rcdecl`
13	`DatatypeCompilation.compile`	`RecordCalc.rcdecl list`	`TypedLambda.tldec`
14	`BitmapCompilation2.compile`	`TypedCalc.tpdecl list`	`BitmapCalc2.bcdec`
15	`ClosureConversion2.convert`	`BitmapCalc2.bcdecl list`	`ClosureCalc.progr`
16	`CallingConventionCompile.compile`	`ClosureCalc.program`	`RuntimeCalc.progr`
17	`ANormalize.compile`	`RuntimeCalc.program`	`ANormal.program`
18	`MachineCodeGen.compile`	`ANormal.program`	`MachineCode.progr`
18	`ConcurrencySupport.insertCheckGC`	`MachineCode.program`	`MachineCode.progr`
19	`StackAllocation.compile`	`MachineCode.program`	`MachineCode.progr`
19	`LLVMGen.compile`	`MachineCode.program`	`LLVMIR.program`
20	`LLVMEmit.emit`	`LLVMIR.program`	`LLVM.LLVMModuleRe`

Part VI

Bibliography and other documents

Bibliography

[1] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–74, 1996.

[2] E. R. Gansner and J. Reppy. The Standard ML Basis Library. Cambridge University Press, 2002.

[3] M.J. Gordon, A.J.R.G. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Lecture Note in Computer Science. Springer-Verlag, 1979.

[4] Lawrence J. Kenah and Simon F. Bate. *VAX/VMS internals and data structures*. Digital Press, Newton, MA, USA, 1984.

[5] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[6] R. Milner, R. Tofte, M. Harper, and D. MacQueen. *The Definition of Standard ML*. The MIT Press, revised edition, 1997.

[7] H-D. Nguyen and A. Ohori. Compiling ml polymporphism with explicit layout bitmap. In *Proceedings of ACM Conference on Principles and Practice of Declarative Programming*, pages 237–248, 2006.

[8] A Ohori. A compilation method for ML-style polymorphic record calculi. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 154–165, 1992.

[9] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995. A preliminary summary appeared at ACM POPL, 1992 under the title "A compilation method for ML-style polymorphic record calculi".

[10] A. Ohori and P. Buneman. Type inference in a database programming language. In *Proc. ACM Conference on LISP and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988.

[11] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli – a polymorphic language with static type inference. In *Proc. the ACM SIGMOD conference*, pages 46–57, Portland, Oregon, May – June 1989.

[12] A Ohori and I. Sasano. Lightweight fusion by fixed point promotion. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 143–154, 2007.

[13] A. Ohori and T. Takamizawa. A polymorphic unboxed calculus as an abstract machine for polymorphic languages. *J. Lisp and Symbolic Comput.*, 10(1):61–91, 1997.

[14] A. Ohori and K. Ueno. Making Standard ML a practical database programming language. In *Proceedings of the ACM International Conference on Functional Programming*, pages 307–319, 2011.

[15] A. Ohori and N. Yoshida. Type inference with rank 1 polymorphism for type-directed compilation of ML. In *Proc. ACM International Conference on Functional Programming*, pages 160–171, 1999.

[16] K. Ueno, A Ohori, and T. Otomo. An efficient non-moving garbage collector for functional languages. In *Proceedings of the ACM International Conference on Functional Programming*, 2011.

[17] Atsushi Ohori, Katsuhiro Ueno, Tomohiro Sasaki, Daisuke Kikuchi. A Calculus with Partially Dynamic Records for Typeful Manipulation of JSON Objects. In *Proceedings of the European Concerence on Object-Oriented Programming*, pages 421-433, 2016.

In *Proc. ECOOP Conference*, pages 18:1-18:25, 2016.

[18] Katsuhiro Ueno, Atsushi Ohori. A fully concurrent garbage collector for functional programs on multicore processors. In *Proceedings of the ACM International Conference on Functional Programming*, pages 421-433, 2016.

[19] 大堀 淳. **プログラミング言語** *Standard ML* **入門**. 共立出版, 2000.