プログラミング言語SML#解説
3.7.1版

大堀 淳　　上野 雄大
東北大学 電気通信研究所

令和3年3月

目 次

第Ⅰ部

概要

第1章　はじめに

本書は東北大学電気通信研究所で開発された関数型プログラミング言語 SML#の公式ドキュメントです．SML#の概要，チュートリアル，SML#言語とライブラリの参照マニュアル，ツールの情報，さらに，SML#の内部構造に関する詳細な記述，等を含む総合ドキュメントを意図しています．

関数型言語に慣れた人はもちろん，これからプログラミングを始めようとする人にも，SML#言語を使って高度なプログラミングを書き始めるために十分な情報を提供します．特に，第 II 部のチュートリアルは，関数型言語の基本的な考え方や ML プログラミングの基礎を含んでおり，ML 言語の手軽な教科書としても使用できます．

SML#は，これら教科書で書かれた Standard ML と後方互換性のある言語です．ネイティブコードコンパイラですが，対話型プログラミングもサポートしており，だれでも手軽に ML プログラミングを楽しむことができます．さらに，SML#が実現している C 言語とのシームレスな連携などの機能は，高度で信頼性の要求される本格的な実用的システムの開発にも威力を発揮すると信じています．

本ドキュメントを参考に，SML#プログラミングをお楽しみください．不明な点や要望等は著者にご連絡ください．

2017 年 6 月

大堀　淳　　上野 雄大
東北大学 電気通信研究所

第2章　本書の構成と執筆状況

本書は以下の部から成ります.

1. 第 I 部: SML#の概要.
2. 第 II 部:SML#のチュートリアル．SML#のインストール方法，SML#の初歩的な使い方，および SML#の拡張機能の概要を，例と共に説明しています．また，関数型言語初学者に向けた，ML プログラミングのための環境のセットアップ方法や，Standard ML の基本機能を用いた ML プログラミングの基本も，本章に含まれています．初心者から上級者までを含む全ての SML#ユーザーにとって，この部の内容は有益なはずです．先頭から順に一気にお読みになることをおすすめします．お急ぎの方は，以下のページにお探しの情報があるかもしれません.
   - **インストール方法：**第 5 章.
   - `smlsharp` **コマンドの起動モードと起動パラメタ：**第 6.5 節.
   - **SML#開発チームと連絡先情報：**第 3.3 節.
   - **SML#の名前について:** 第 3.2 節.
   - **SML#ホームページ：**`https://smlsharp.github.io/`
   - **SML# Document in English：**`https://smlsharp.github.io/en/documents/3.7.1/`
3. 第 III 部: SML#言語およびライブラリの参照マニュアル.

   SML#言語の構文と意味の定義，および，SML#プログラミングに使用する Standard ML 基本ライブラリと SML#の機能を活用する種々のライブラリの API と機能の定義を含みます.
4. 第 IV 部: SML#付属のツール群（`smllex`, `smlyacc`, `smlformat` など）の解説と参照マニュアル.
5. 第 V 部:SML#コンパイラの内部構造．SML#コンパイラの開発や関数型言語のコンパイラの構築方法に興味のある方を対象に，ソースコードとデータ構造の詳細を記述します.
6. 第 VI 部: 参考文献のリスト.

このうち，3.7.1 版では第 IV 部および第 V 部は執筆途中です．将来の版での完成を目指しています.

第3章　SML#の概要

本章ではSML#言語の概要を説明します．

3.1　SML#とは？

SML#は，以下のような特徴をもったML系関数型プログラミング言語です．

1. **Standard MLとの後方互換性．**SML#は，ML系言語の標準の厳密な仕様であるStandard MLとの後方互換性を持っています．わずかな例外を除いて，Standard MLの形式的な仕様[5]を満たすすべてのプログラムをコンパイルできます．

2. **レコード多相性．**レコード多相性[9]は，オブジェクトやデータベースのタプルなどに現れるラベル付きレコードをML言語に完全に統合するために必要な型システムの機能です．SML#は，この機能を完全にサポートしています．

3. **SQLのシームレスな統合．**SQLはデータベースの標準問い合わせ言語であり，データベースを利用するプログラムで必ず必要となる機能です．SML#は，SQLの一部の機能をライブラリとして提供するのではなく，SQLクエリそのものを多相型をもつ第一級の式として統合しています．この機能により，複雑なプログラムデータベース操作を，MLプログラムの中で直接プログラムすることができます．

4. **C言語との直接連携．**システムプログラミングを含む種々のOSの機能の利用にはC言語で記述されたライブラリへのアクセスが必要となります．SML#では，名前を外部名宣言するだけで，C言語で書かれCコンパイラでコンパイルされた関数を呼び出すことができます．

5. **マルチコアCPU上のネイティブスレッドのサポート．**SML#の並行かつオブジェクトを動かさないGCの機能により，C言語との連携機能を使い，OSのスレッドライブラリを直接呼び出すことができます．従って，OSがマルチコアCPU上での並列実行をサポートしさえすれば，スレッドを使った高水準なMLプログラムを書き，マルチコアCPU上で効率良く実行することができます．

6. **分割コンパイルとリンク．**SML#は，従来のインクリメンタルなコンパイルではない，真の分割コンパイルを実現しています．各モジュールのインターフェイスの確定後は，各モジュールを独立に開発しコンパイルやテストできます．さらに，SML#コンパイラは，分割コンパイルの対象となる各ソースコードを，システム標準のオブジェクトファイル（例えばLinuxならばELFフォーマット）にコンパイルし，システムのリンカーでCのライブラリなどとともにリンクします．この機能により，Standard MLだけでなくC言語やSQLなどをも使う大規模プログラムを安全かつ効率的に開発することができます．

SML#コンパイラおよび実行時処理系は，東北大学電気通信研究所大堀・上野研究室で開発され，東北大学が著作権保有するオープンソースソフトウェアです．BSDスタイルのSML#ライセンス（4節参照）によって公開されており，だれでも自由に利用することができます．このライセンスは，二次著作物（つまりコンパイラで作成されたシステムなど）に関する制限の少ないフリーソフトウェアライセンスであり，企業の商品開発にも安心して使用できます．

3.2　SML#の歴史

SML#の起源は，大堀等によるデータベースコミュニティへの Machiavelli 言語の提案 [11] に遡ることができます．この論文では，レコード多相を含む ML は SQL を包摂できることが示され，その有用性がプロトタイプの実装によってしめされています．

その後，1993 年，沖電気工業（株）関西総合研究所にて，大堀により，Standard ML of New Jersey コンパイラに多相型レコード演算を加え拡張したプロトタイプ **SML# of Kansai** が開発されました．その時の SML#の types メーリングリストへのアナウンスは，今でもインターネット上に記録されています (`http://www.funet.fi/pub/languages/ml/sml%23/description`).

SML# of Kansai の名前は，このコンパイラにて初めて多相型が与えられたレコード演算子"`#`"を象徴するものです．このコンパイラは，1996 年 ACM TOPLAS に出版されたコード多相性に関する論文 [9] で，**SML#**の名前で紹介されています．

その後，より完全な SML#コンパイラの開発の努力が継続されました．

2003 年に，北陸先端科学技術大学院大学にて，文部科学省リーディングプロジェクト e-Society 基盤ソフトウェアの総合開発「高い生産性をもつ高信頼ソフトウエア作成技術の開発」（領域代表者：片山卓也）の一つの課題「プログラムの自動解析に基づく高信頼ソフトウェアシステム構築技術」（2003 年―2008 年，究代表者:大堀 淳）（`http://www.tkl.iis.u-tokyo.ac.jp/e-society/index.html` ）として，次世代 ML 系関数型言語 SML#をスクラッチから開発するプロジェクトを開始しました．

2006 年 4 月，プロジェクトは，大堀とともに東北大学電気通信研究所に移り開発を継続しています．

2008 年の e-Society プロジェクト終了後も，東北大学電気通信研究所大堀・上野研究室にて，SML#の研究開発を続けています．

3.3　SML#開発チームと連絡先情報

現在（2021-03-15）SML#は，

- 大堀 淳（東北大学電気通信研究所）
- 上野雄大（東北大学電気通信研究所）

の 2 名が，東北大学大学院情報科学研究科の大堀・上野研究室に所属の大学院生の協力の下，開発を行なっています．

SML#のこれまでの主な開発者は以下の通りです．(敬称は略させていただいています．所属は SML#の開発に携わった時のものです．)

- 大堀 淳（北陸先端科学技術大学院大学情報科学研究科，東北大学電気通信研究所）
- 大和谷 潔（算譜工房）
- Nguyen Huu Duc（北陸先端科学技術大学院大学情報科学研究科，東北大学電気通信研究所）
- Liu Bochao（北陸先端科学技術大学院大学情報科学研究科，東北大学電気通信研究所）
- 纓坂 智（北陸先端科学技術大学院大学情報科学研究科）
- 上野雄大（北陸先端科学技術大学院大学情報科学研究科，東北大学電気通信研究所）

東北大学電気通信研究所では，SML#に関する情報共有の目的で以下の web サイトやメーリングリストを管理しています．

- SML#ホームページ. `https://smlsharp.github.io/` コンパイラや本書を含む文書の最新版などもここからダウンロードできます．

- Twitter アカウント `@smlsharp`．SML#に関する最新情報をツイートします．

  修正・執筆にあたっては，twitter などでの SML#や本書へのコメントなども，本書の改善の参考にさせて頂いております．個々に言及いたしませんが，本書をご覧頂きコメントをお寄せ頂いた方々に感謝いたします．

3.4 謝辞

2003 年にスタートした SML#開発の過程では，以下を含む色々なご指導やご協力を頂きました．ここに謝意を表します．

3.4.1 プロジェクトファンディング

SML#言語の研究開発は，2003 年から 5 年間の文部科学省リーディングプロジェクト e-Society 基盤ソフトウェアの総合開発「高い生産性をもつ高信頼ソフトウェア作成技術の開発」(`http://www.tkl.iis.u-tokyo.ac.jp/e-society/index.html`) の一つの課題「プログラムの自動解析に基づく高信頼ソフトウェアシステム構築技術」(究代表者:大堀 淳) としてスタートをきることができました．このプロジェクトの主要な目標が SML#言語コンパイラの開発でした．SML#は開発ソースの総量が 30 万行を超える大規模システムです．このプロジェクトの支援がなければ，SML#の開発は困難であったと思われます．文部科学省，e-Society の領域代表の片山卓也先生，および関係各位に深謝いたします．この大型プロジェクトに加え，SML#の理論や実装方式の研究開発には，複数の科学研究費補助金によるサポートを受けています．

3.4.2 SML#が使用しているソフトウエア

SML#言語は，第 1.0 版以降は，SML#自身でコンパイルし開発を行なっていますが，それ以前は，Standard ML of New Jersey および MLton の Standard ML コンパイラを使って開発を行いました．

SML#は前節（3.3）の開発チームによって開発されたソフトウェアです．ソースコードの殆どをスクラッチから開発しましたが，一部に以下のコードを利用しています．

内容	SML#ソース上の位置	ソース
ML-Yacc	src/ml-yacc	Standard ML of New Jersey 110.73
ML-Lex	src/ml-lex	Standard ML of New Jersey 110.73
ML-Lpt	src/ml-lpt	Standard ML of New Jersey 110.99
SML/NJ Library	src/smlnj-lib	Standard ML of New Jersey 110.99
TextIO, BinIO, OS, Date, Timer	src/smlnj	Standard ML of New Jersey 110.73
浮動小数点／文字列変換関数	src/runtime/netlib	the Netlib

これらソースは，いずれも SML#ライセンスと整合性あるライセンスで配布されているオープンソースソフトウェアです．「SML#ソース上の位置」およびソースパッケージのトップレベルにそれぞれのライセンスが添付されています．

3.4.3 研究開発協力者

SML#言語の開発にあたっては，多くの人々からの指導を受けました．過去現在の開発チーム（第（3.3）節）以外で特に貢献のあった方々は以下の通りです．

- 篠埜功氏．大堀と共に，関数フュージョン機能を持つ新たしいインラインの理論およびその実験的な実装を行いました．この機能は実験的な実装がありますが，まだ十分な完成度が得られておらずSML#3.7.1 版に組み込まれていませんが，将来取り入れたいと考えています．
- 大友聡顕氏．大堀，上野と共に，オブジェクトを動かさない GC の研究開発に携わり，初期の実験的な実装を行い，この方式が有望であることを確認しました．この成果は，現在のオブジェクトを動かさない GC の方式の開発と実装の契機となったものです．

これらの方々以外にも，SML#は，大堀等との共同で行なった様々な型理論やコンパイル方式の基礎研究を基に設計されています．現在の SML#言語に直接生かされている出版された基礎研究成果には以下のものが含まれます．

- レコード多相性の理論 [8, 9].
- データベースの型推論 [10].
- データベース言語（Machiavelli）[11, 1].
- ランク 1 多相性の型理論 [15].
- ML の unboxed 意味論 [13].
- ML における自然なデータ表現 [7].
- 軽量の関数融合 [12].
- オブジェクトを動かさない GC[16].
- SQL の統合方式 [14].
- JSON サポート [17].
- 並行 GC[18].

それ以外にも多くの共同研究者から色々な機会に示唆や助言を頂いており，それらは 1989 年にまで遡りますが，それらの方々の列挙は割愛させていただきます．

3.5　SML#第 3.7.1 版の機能と制限

我々開発チームは第 3.1 節でのべた機能をすべて開発し SML#開発開始時に目標とした機能を実現しています．第 3.7.1 版にその殆どが含まれていますが，以下の制約があります．

1. ターゲットアーキテクチャ．現在の SML#コンパイラは，Intel アーキテクチャ（x86_64）向けのコードのみ生成可能です．将来，マルチターゲット化を行う予定です．
2. 最適化．現在のバージョンには，インライニングや定数の伝播などの基本的な最適化も十分に実装されていません．従って，コンパイル時間，コンパイルされたコードの実行時間もともに十分とは言えません．将来の版では，他の最適化コンパイラに比肩する速度が得られると期待しています．

第4章　SML#ライセンス

第II部

チュートリアル

第5章　SML#のインストール

5.1　動作環境

第II部では，SML#でプログラミングをマスターするためのチュートリアルを提供します．まずはSML#コンパイラをインストールし，SML#プログラミング環境を整えましょう．

SML#3.7.1 版は以下のプラットフォームで動作します．

- Linux（amd64（x86_64）版）
- macOS（10.15 以降推奨）
- Windows 10（Windows Subsystem for Linux を利用）

また，SML#のコンパイルと実行には以下のサードパーティ製ソフトウェアが必要です．

- MassiveThreads 1.00
- GNU Multiple Precision Arithmetic Library (GMP) ライブラリ
- LLVM 3.9.1 以降（LLVM 11.0.0 を推奨．新しいものが望ましい）

MassiveThreads は BSD スタイルライセンスの下で配布されているフリーソフトウェアです．GMP は LGPL（GNU Lesser General Public License）の下で配布されているフリーソフトウェアです．LLVM は例外条項付きの Apache License version 2 の下で配布されているフリーソフトウェアです．

LLVM はバージョン 3.9.1 以降のできるだけ新しいものをご用意ください．LLVM 11.0.0 までのバージョンでコンパイルと動作を確認済です．3.8 以前の LLVM では SML#をビルドすることができません．

これらのソフトウェアは全て，SML#コンパイラのビルドや実行に必要です．以下の点に注意してください．

- SML#コンパイラのビルドには，これら全てのライブラリと `opt`，`llc`，および `llvm-as` コマンドが使用されます．SML#コンパイラコマンドには MassiveThreads および GMP ライブラリがリンクされます．LLVM のライブラリはリンクされません．
- インストール後の SML#コンパイラコマンドは，コード生成のために `opt`，`llc`，`llvm-as`，および `llvm-dis` コマンドを使用します．
- SML#コンパイラが生成したすべての実行形式ファイルには，MassiveThreads および GMP ライブラリがリンクされます（これらのライブラリの機能を使用していなくても，これらのライブラリがリンクされます．この点は将来改善する予定です）．

これらのライブラリおよびコマンドは，SML#の配布パッケージには含まれません．SML#コンパイラをインストールするユーザは，これらのライブラリを別途インストールする必要があります．多くの場合，OS のパッケージ管理システムなどで簡単に導入できるはずです．

以上の環境があれば，SML#は，簡単な手順でインストールできます．各 OS 毎のインストール処理の詳細は以下の各節のとおりです．

5.2 Debian GNU/Linux

SML#リリース時点で最新リリースの Debian GNU/Linux および Debian sid に対してプライベートリポジトリを用意しています．このリポジトリをシステムに追加することで，apt コマンドで SML#コンパイラのインストールおよびアップデートを行うことができます．

以下のコマンドでインストールできます．

- Debian sid:

```
wget -P /usr/share/keyrings https://github.com/smlsharp/repos/raw/main/debian/dists/sid/sml
wget -P /etc/apt/sources.list.d https://github.com/smlsharp/repos/raw/main/debian/dists/sid
apt update
apt install smlsharp
```

- Debian 10 (buster):

```
wget -P /usr/share/keyrings https://github.com/smlsharp/repos/raw/main/debian/dists/buster/
wget -P /etc/apt/sources.list.d https://github.com/smlsharp/repos/raw/main/debian/dists/bus
apt update
apt install smlsharp
```

詳細は，以下のとおりです（sid の場合）．

1. SML#開発チームの公開鍵をダウンロードしシステムの所定の位置に置きます．

   ```
   wget -P /usr/share/keyrings https://github.com/smlsharp/repos/raw/main/debian/dists/sid/sml
   ```

   以下のコマンドで鍵指紋を確認できます．

   ```
   gpg --with-fingerprint /usr/share/keyrings/smlsharp-archive-keyring.gpg
   ```

   ダウンロードした鍵の指紋が上記指紋と一致することを確認してください（より厳密に鍵の正当性を確認したい場合は，SML#開発者本人に直接会って鍵指紋を受け取ってください）．鍵指紋は以下の通りです．

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

2. プライベートリポジトリの記述ファイルをダウンロードし，システムに追加します．

   ```
   wget -P /etc/apt/sources.list.d https://github.com/smlsharp/repos/raw/main/debian/dists/sid
   ```

3. プライベートリポジトリからパッケージ一覧を取得します．

   ```
   apt update
   ```

4. SML#コンパイラをインストールします．LLVM や MassiveThreads などの依存するライブラリも必要に応じてインストールされます．SMLFormat などのツールもおすすめパッケージとしてインストールされます．

   ```
   apt install smlsharp
   ```

5.3 Ubuntu

SML#リリース時点で最新のリリースおよび LTS リリースに対して PPA (Personal Package Archives for Ubuntu) にプライベートリポジトリをを用意しています．このリポジトリをシステムに追加することで，apt コマンドで SML#コンパイラをインストール，アップデートすることができます．
以下のコマンドでインストールできます．

- Ubuntu 20.10 LTS (Groovy):

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

- Ubuntu 20.04 LTS (Focal):

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

add-apt-repository でシステムにプライベートリポジトリを追加します．apt コマンドは Debian の場合と同様です．詳細は 5.2 節をご覧ください．

5.4 Fedora

SML#リリース時点で最新リリースの Fedora および Fedora Rawhide に対してプライベートリポジトリを用意しています．このリポジトリをシステムに追加することで，dnf コマンドで SML#コンパイラのインストールおよびアップデートを行うことができます．
以下のコマンドでインストールできます．

- Fedora:

```
rpm -i https://github.com/smlsharp/repos/raw/main/fedora/smlsharp-release-fedora-31-0.noarc
dnf install smlsharp smlsharp-smlformat smlsharp-smllex smlsharp-smlyacc
```

- Fedora Rawhide:

```
rpm -i https://github.com/smlsharp/repos/raw/main/fedora/smlsharp-release-rawhide-31-0.noar
dnf install smlsharp smlsharp-smlformat smlsharp-smllex smlsharp-smlyacc
```

詳細は，以下のとおりです（Rawhide の場合）．

1. SML#開発チームの公開鍵およびプライベートリポジトリの設定ファイルを含む RPM パッケージをダウンロードしインストールします．

   ```
   rpm -i https://github.com/smlsharp/repos/raw/main/fedora/smlsharp-release-fedora-31-0.noarc
   ```

2. SML#コンパイラおよび周辺ツールをインストールします．

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

   dnf コマンドの実行中，SML#開発チームの公開鍵をインポートすることについて何度か聞かれます．鍵指紋を確認の上，許可してください．鍵指紋は以下の通りです．

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

5.5 CentOS

CentOS 7 および CentOS 8 に対して，amd64 用のプライベートリポジトリを用意しています．このリポジトリをシステムに追加することで，`yum` コマンドや `dnf` コマンドで SML#コンパイラのインストールおよびアップデートを行うことができます．
以下のコマンドでインストールできます．

- CentOS 8:

```
rpm -i https://github.com/smlsharp/repos/raw/main/centos/smlsharp-release-centos-8-0.noarch
dnf install smlsharp smlsharp-smlformat smlsharp-smllex smlsharp-smlyacc
```

- CentOS 7:

```
yum install epel-release
rpm -i https://github.com/smlsharp/repos/raw/main/centos/smlsharp-release-centos-7-0.noarch
yum install smlsharp smlsharp-smlformat smlsharp-smllex smlsharp-smlyacc
```

コマンドは，以下の点を除いて Fedora の場合とほぼ同様です．

- CentOS 7 の場合，`epel-release` パッケージを事前にインストールしておく必要があります．

その他の詳細は 5.4 節をご覧ください．

5.6 macOS

Homebrew を使ってインストールするのが最も簡単です．Homebrew をセットアップした後，以下のコマンドを実行し，必要なライブラリと SML#をインストールしてください．

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

詳細は，以下のとおりです．

1. `http://brew.sh/` を参照し，Homebrew をセットアップします．
2. SML#開発チームが提供する git リポジトリを tap し，SML#関連パッケージを Homebrew システムに加えます．

   ```
   brew tap smlsharp/smlsharp
   ```

   このリポジトリは MassiveThreads と SML#コンパイラの formula（パッケージ）および bottle（バイナリパッケージ）を含みます．ただし，bottle は SML#リリース時の最新の macOS に対してのみ提供されます．パッケージの正当性を示すため，このリポジトリの各コミットは SML#開発チームの秘密鍵で署名されています．署名を検証したい場合は以下の URL あるいは PGP 公開鍵サーバーから SML#開発チームの公開鍵を取得してください．

   ```
   https://github.com/smlsharp/repos/raw/main/debian/dists/sid/smlsharp-archive-keyring.gpg
   ```

   鍵指紋は以下の通りです．

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

3. SML#コンパイラをインストールします．

```
brew install smlsharp
```

LLVM や MassiveThreads なども自動でインストールされます．Bottle が用意されているバージョンの macOS を使用していない場合は，MassiveThreads と SML#はソースからビルドされるため，このコマンドの完了には長い時間がかかります．

5.7 Windows 10

Windows Subsystems for Linux と Ubuntu をセットアップすることで，SML#の Ubuntu 用バイナリパッケージを Windows にそのままインストールし実行することができます．

Windows Subsystems for Linux を有効にするための手順の詳細は，Microsoft が提供するドキュメントをご覧ください．bash を起動したあとのインストール手順は Ubuntu へのインストール手順と同一です (5.3 節をご覧ください)．

5.8 ソースからビルドする場合

その他のシステムではソースからビルドしてください．ソースからのビルドには，以下の開発ツールとライブラリが事前にインストールされている必要があります．

1. プログラム開発用のツール群 GNU binutils（GNU Binary Utilities）．
2. C および C++コンパイラ（gcc または clang）
3. make（GNU make を推奨）
4. MassiveThreads および GMP ライブラリ本体とヘッダーファイル
5. LLVM 3.9.1 以降のライブラリ，ヘッダーファイル，およびコマンド

これらのソフトウェアがインストール済みならば，SML#は標準的な 3 ステップ `./configure && make && make install` でインストールすることができます．

これらのソフトウェアが OS のパッケージシステムによって提供されていない場合，これらもソースからビルドする必要があります．これらのコンパイルの詳細は，各ソフトウェアの公式ドキュメントなどを参照してください．

本文書の執筆時点での MassiveThreads および LLVM 11.0.0 の大まかなインストール手順は以下の通りです．

MassiveThreads MassiveThreads の Web ページ `https://github.com/massivethreads/massivethreads` などから MassiveThreads 1.00 のソースコード `massivethreads-1.00.tar.gz` を入手，展開し，標準的な 3 ステップ `./configure && make && make install` でインストールしてください．

LLVM 11.0.0 LLVM の Web ページ `http://llvm.org/` などから LLVM のソースコード `llvm-11.0.0.src.tar.xz` を入手，展開し，以下の 5 ステップでビルドします．

```
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
```

上記例で cmake に指定しているオプションは必須ではありませんが，指定することをおすすめします．-DCMAKE_INSTALL_PREFIX は他の LLVM のインストールと衝突を避けるために指定することをおすすめします．-DCMAKE_BUILD_TYPE=Release は LLVM 自体を最適化します．-DLLVM_BUILD_LLVM_DYLIB=On オプションは共有ライブラリをビルドします．-DLLVM_ENABLE_BINDINGS=Off は SML#に不要なモジュールのビルドを防ぎます．

以上の環境の下で，以下の手順で SML#をソースからビルドします．

1. ソースパッケージ（https://github.com/smlsharp/smlsharp/releases/download/v3.7.1/smlsharp-3.7.1.tar.gz）をダウンロードします．最新のソースパッケージは https://smlsharp.github.io/ja/downloads/ からも取得できます．

2. 適当な SML#ソースディレクトリを決め，そこにソースパッケージを展開します．ただし，絶対パスに日本語が含まれるディレクトリは避けてください．SML#をビルドできません．

3. 適当な SML#のインストール先ディレクトリを決めます．以下，そのディレクトリを *prefix* とします．

4. SML#ソースディレクトリにて configure スクリプトを実行します．このとき，--prefix オプションに *prefix* を，--with-llvm に LLVM をインストールしたディレクトリを指定します．

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

   --prefix オプションを省略した場合は /usr/local にインストールされます．--with-llvm オプションを省略した場合は，LLVM のライブラリとコマンドを標準的な場所（/usr/bin など）から探します．

5. make コマンドを実行しビルドを行います．

   ```
   $ make
   ```

   なお，ビルド終了後，以下のコマンドで SML#コンパイラをインストールする前に起動することができます．

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

6. （オプション）以下のコマンドで，ビルドした SML#コンパイラで SML#コンパイラをビルドし直すことができます．

   ```
   $ make stage
   $ make
   ```

7. make install コマンドでインストールします．

   ```
   $ make install
   ```

   システム管理の都合上，インストールされるファイルを *prefix* とは異なるディレクトリ *prefix*′ に出力させたい場合は，make install コマンドに DESTDIR=*prefix*′ オプションを指定してください．

以上により，以下のファイルがインストールされます．

1. smlsharp コマンド：*prefix*/bin/smlsharp

2. smlformat コマンド：*prefix*/bin/smlformat

3. `smllex` コマンド：*prefix*`/bin/smllex`
4. `smlyacc` コマンド：*prefix*`/bin/smlyacc`
5. ライブラリファイル：*prefix*`/lib/smlsharp/`ディレクトリ以下のファイル

以下のコマンドでSML#コンパイラが起動できるはずです．

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

補足とヒント：

- マルチコアCPUをお使いの方は，GNU makeならば`make`コマンドを実行するとき`-j` $\langle n \rangle$ スイッチを指定すると，並列にコンパイルされ，実行時間が短縮される場合があります．$\langle n \rangle$ は並列度です．コアの数程度に指定すると良い結果が得られると思います．

第6章　SML#プログラミング環境の準備

6.1　Unix系OS，Emacsエディタ，その他ツールの整備

プログラミングのためには，

- 使いやすい高機能エディタ
- コンパイラとリンカー

を含むプログラミング環境が必要です．SML#でのプログラミングに必要とされるプログラミング環境は，SML#コンパイラのインストールを除けば，C 言語の場合と同様です．Java などの言語では，これらを統合した対象言語に特化した Eclipse などの統合開発環境を使用する場合が多いですが，C との直接連携機能を用いたシステムプログラミングや SQL を使ったデータベース操作などの SML#の先端機能を駆使したプログラミングを十分に楽しむためには，以下のような標準的なプログラミング環境を整えることをお勧めします．

- **Unix 系の OS.** 高度なプログラム開発には，Linux，FreeBSD，Mac OS X 等の Unix 系 OS が豊富なツールを含んでいて便利です．Windows 系の OS がインストールされた PC の場合は，VMware，VirtualBox などの仮想マシンを利用して Linux などの環境を容易に構築できます．

- **Emacs エディタ.** Emacs はプログラミングにもっとも適したエディタの一つです．プログラミングは，テキスト形式の文を書き，コンパイルしテストをし，修正する，という作業の繰り返しであり，その大部分の時間はテキストエディタとの付き合いとなります．そこで，強力でカスタマイズ可能なテキストエディタの選択が重要です．Emacs 系テキストエディタ（GNU Emacs，XEmacs など）は，複数のバッファの操作，ディレクトリなどのファイルの管理，コンパイラなどのコマンド起動，さらに，それらの柔軟なカスタマイズ機能を備えた強力なエディタであり，プログラミングや LaTeX 文書などの複雑で高度な文書の作成に最適なエディタです．使い始める時に少々練習が必要ですが，使いこなせば，プログラム開発の強力な道具となります．Unix 系 OS では通常，標準で用意されています．

- **C コンパイラ.** SML#コンパイラは，ソースコードを x86_64 アーキテクチャのネイティブコードにコンパイルし，OS 標準のオブジェクトファイルを作成し，さらにそれらをリンクし実行形式ファイルを作成します．この過程で C コンパイラドライバ（`gcc` や `clang` など）を通してリンカーを呼び出します．C コンパイラは，Unix 系 OS のインストール時にプログラム開発用に適したシステムを選択すれば標準でインストールされているはずです．

  SML#プログラムのみをコンパイルしリンクする場合は，ユーザが直接 C コンパイラを呼び出す必要はありませんが，C 言語との直接連携機能を使いより高度なプログラミングを行うためにも，C コンパイラに慣れておくことをお勧めします．

- **データベースシステム.** SML#のデータベース機能を使用するにはデータベースシステムをインストールする必要があります．第 3.7.1 版は PostgreSQL，MySQL，ODBC，SQLite3 に対応しています．データベースアクセス機能を使用するには，これらのうちいずれかのデータベースシステムをインストールする必要があります．これらのうち，SML#との連携が最もテストされているのは PostgreSQL です．詳細は OS に依存しますが，いずれの場合も簡単にインストールできるはずです．

6.2　SML#コンパイラの構造とブートストラップ

この節の内容は，SML#コンパイラをインストールし使用する上で理解する必要はありませんが，やや時間がかかる SML#コンパイラのインストール処理の理解や，さらにコンパイラの一般的な構造を理解する上で有用と思います．

SML#3.7.1 版コンパイラは，SML#言語で書かれたファイルを分割コンパイルし，ネイティブコードを生成します．対話型モードも，このコンパイラを使い，(1) 現在の環境下で分割コンパイル，(2) システムとのリンク，(3) オブジェクトファイルの動的ロードを繰り返すことによって実現しています．SML#コンパイラは SML#言語，C 言語，および C++言語で書かれており，さらに自分自身のコンパイル時に以下のツールを使用しています．

- ml-lex，ml-yacc．字句解析および構文解析ツール．
- SMLFormat．プリンタ自動生成ツール
- 基本ライブラリ

これらも SML#言語で書かれています．

SML#コンパイラは，各 SML#ファイル（Standard ML ファイルを含む）をシステム標準のオブジェクトファイルにコンパイルします．コンパイルしたファイルは，システム標準のリンカによって実行形式ファイルに結合できます．従って，SML#コンパイラは，C/C++コンパイラと SML# コンパイラがあれば構築できます．しかし，もちろん，SML#コンパイラを初めてインストールする時には，SML#コンパイラはまだ存在しません．SML#3.7.1 版では，以下の手順でこのブートストラップの問題を解決し，SML#をビルドしインストールしています．

1. C/C++で書かれたランタイムライブラリをコンパイルし静的リンクライブラリを生成します．
2. SML#コンパイラのソースコードをコンパイルするのに十分な大きさの SML#コンパイラ（`minismlsharp`）を，古い SML#コンパイラで事前にコンパイルしておき，SML#のソースツリーにコンパイル済みの LLVM IR ファイルを用意しています．
3. インストール先のシステムの下で，`minismlsharp` の LLVM IR ファイルを LLVM を用いてコンパイルし，ランタイムライブラリとリンクし，`minismlsharp` コマンドを生成します．
4. `minismlsharp` コマンドを用いて，SML#をコンパイルするための各種ツールとライブラリ，および SML#コンパイラをコンパイルします．このコンパイルは，ツールとライブラリの依存関係から，おおよそ以下の順番で行われます．
   (a) 基本ライブラリ（Basis Library）のコンパイル．
   (b) `smllex` コマンドのコンパイルとリンク．
   (c) ML-yacc ライブラリと `smlyacc` コマンドのコンパイルとリンク．
   (d) `smllex` および `smlyacc` コマンドによるパーザの生成．
   (e) `smlformat` コマンドのコンパイルとリンク．
   (f) `smlformat` コマンドによるプリンタの生成．
   (g) すべてのライブラリのコンパイル．
   (h) `smlsharp` コマンドのコンパイルとリンク．
5. 以下のファイルを所定の位置にインストールします．
   - ランタイムライブラリのライブラリファイル．

- ライブラリのインターフェースファイル，オブジェクトファイル，およびシグネチャファイル．
- `smllex`，`smlyacc`，`smlformat`，`smlsharp` の各コマンド．

これらは以上のステップが示す様に，各ソースファイルやコマンド間には複雑な依存関係があります．さらに，いくつかのソースファイルの処理は，OS の環境に依存しています．これらは，大規模なシステムの典型です．これら依存関係を制御する一つの手法は，GNU Autoconf による `configure` スクリプトと `make` コマンドの使用です．SML#コンパイラは，インターフェイスファイルに従って各ファイルをコンパイルします．個別のファイルが必要とするファイルは，このインターフェイスファイルに記述されています．SML#コンパイラは，この依存関係を `make` コマンドが読み込めるフォーマットで出力する機能を持っています．

1. `smlsharp -MM` *smlFile*．ソースファイル *smlFile* を読み，このソースファイルをコンパイルするために必要なファイルのリストを Makefile 形式で出力します．
2. `smlsharp -MMl` *smiFile*．インターフェイスファイル *smiFile* をトップレベルのファイルとみなし，このトップレベルから生成されるべき実行形式コマンドのリンクに必要なオブジェクトファイルのリストを Makefile 形式で出力します．
3. `smlsharp -MMm` *smiFile*．インターフェイスファイル *smiFile* をトップレベルのファイルとみなし，このトップレベルから生成されるべき実行形式ファイルをビルドするための Makefile を生成します．

SML#システムは，この機能を利用して，上記の手順を実行する `Makefile` を生成しています．この `Makefile` によって，通常の大規模システムの開発と同様，再コンパイルが必要なファイルのみ `make` コマンドによってコンパイル，リンクが実行されます．

6.3 SML#の対話型モードを使ってみよう

インストールが終了すると，SML#コンパイラが `smlsharp` という名前のコマンドとして使用可能となります．このコマンドをコマンドシェルや Emacs の shell モードから起動することによって，SML#プログラムをコンパイル，実行できます．もっとも簡単な使用法は，SML#コンパイラを対話モードで実行することです．`smlsharp` を引数なしで起動すると，対話モードの起動要求とみなし，起動メッセージを表示しユーザからの入力待ちの状態となります．

```
$ smlsharp
SML# version 3.7.1 (2021-03-15) for x86_64-pc-linux-gnu with LLVM 11.0.0
#
```

「`# `」は SML#システムが印字するプロンプト文字です．またこの文書では，シェル（コマンド）のプロンプトを「`$ `」と仮定します．

この後，コンパイラは以下の処理を繰り返します．

1. 区切り文字; まで，端末からでプログラムを読み込む．
2. 読み込んだプログラムをコンパイルし，プログラムを呼び出し実行する．
3. プログラムが返す結果を表示する．

以下に簡単な実行例を示します．

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

最初の行がユーザの入力です．2 行目が，ユーザの入力に対する SML#システムの応答です．この例のように，コンパイラは，プログラムの実行結果に加えコンパイラが推論した型を表示します．さらに，実行結果に名前をつけ，これに続くセッションで利用可能にします．ユーザによる指定がなければ，it という名前が付けられます．

6.4　SML#のコンパイルモードを試してみよう

SML#コンパイラは，対話型以外に，C コンパイラのようにファイルをコンパイルすることができます．

まず，簡単な例として，前節での入力`# "Hello world";`
をファイル書いてコンパイルしてみましょう．そのために，`hello1.sml` を以下のような内容で作成します．

```
"Hello world";
```

最後のセミコロンはあってもなくても同じです．この作成したファイルは，以下のようにしてコンパイルできます．

```
$ 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]=43a2c24d3728ad6a35
not stripped
$
```

SML#は，引数としてファイル名のみが与えられると，C コンパイラ同様，そのファイルをコンパイル，リンクし実行形式ファイルを作成します．以上のようにシステム標準の実行形式ファイル `a.out` が作成されていることが確認できます．作成されるファイル名は tt -o スイッチで指定できます．

```
$ smlsharp -o hello1 hello1.sml
$ ls hello1*
hello1 hellp1.sml
$
```

作成したファイルは，通常のコマンドと同じように実行できます．

```
$ ./hello1
$
```

何も表示されませんが，これで正常です．ファイルのコンパイルでは，対話型モードと違い，結果の値と型を表示するコードは付加されません．

結果を表示したければ，結果を表示する関数を呼ぶ必要があります．そこで，`hello2.sml` を以下のような内容で作成します．

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

このファイルには，このファイルで定義していない `print` 関数が使用されています．我々の意図は基本ライブラリで定義された `print : string -> unit` を使用することですが，`print` は自由に再定義できますから，SML#コンパイラにとっては，名前 `print` が基本ライブラリ関数の `print` を指すとはかぎりません．そこでこのファイルをコンパイルするためには，`print` が定義されているファイルをコンパイラに通知する必要があります．

このようにファイルに定義された名前以外の名前を使用するファイルをコンパイルするためには，そのファイルで使用する名前が他のソースファイルで定義されていることを，**インターフェイスファイル**で宣

言する必要があります．SML#は，ソースファイル名に対応するインターフェイスファイル"hello2.smi"を探します．hello.sml を実行するためには，以下の"hello2.smi"ファイルを以下のように作成する必要があります．

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

この宣言は，hello2.sml がインターフェイスファイル"basis.smi"で宣言されている資源を必要としていることを表しています．"basis.smi"は Standard ML の基本ライブラリで定義されているすべての名前が宣言されているインターフェイスファイルです．

この用意の下で，以下のコマンドを実行すれば，実行形式プログラムが作成されます．

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

6.5 smlsharp コマンドの起動モード

smlsharp の主な実行モードは以下の 4 つです．

対話型モード smlsharp をパラメタなしに起動すると，対話型セッションを実行します．

コンパイル・リンクモード 一つのソースファイルを指定して起動するとそのファイルをコンパイルし，実行形式プログラムを作成します．

コンパイルモード -c スイッチを指定して起動すると，指定されたソースファイルをオブジェクトファイルにコンパイルします．

リンクモード 一つのインターフェイスファイル（smi ファイル）を指定して起動すると，インターフェイスファイルに対応するソースファイルとインターフェイスで参照されたすべてのソースファイルがコンパイル済みとみなし，それらすべてのオブジェクトファイルをリンクし，実行形式プログラムを作成します．

これら実行を制御する起動スイッチとパラメタには以下のものが含まれます．

--help ヘルプメッセージをプリントし終了．

-v 種々のメッセージを表示．

-o ⟨*file*⟩ 出力ファイル（オブジェクトファイル，実行形式ファイル）の名前を指定．

-c コンパイルしオブジェクトファイルを生成．

-S コンパイルしアセンブリファイルを作成．

-M コンパイルの依存関係を表示

-MM システムライブラリを除いたコンパイルの依存関係を表示

-Ml リンクの依存関係を表示

-MMl システムライブラリを除いたリンクの依存関係を表示

-Mm Makefile を生成

-MMm システムライブラリを除いた Makefile を生成

`-I` ⟨*dir*⟩ ソースファイルのサーチパスの追加

`-L` ⟨*dir*⟩ リンカーへのライブラリファイルのサーチパスの追加

`-l` ⟨*libname*⟩ リンク時にライブラリ ⟨*libname*⟩ をリンク

`-Wl,`⟨*args*⟩ リンク時に C コンパイラドライバ（`gcc` や `clang`）へ ⟨*args*⟩ をパラメタとして渡す

`-c++` リンク時に用いるコンパイラドライバとして C++コンパイラを用いる．C++で書かれたライブラリと SML#プログラムをリンクするときに指定します．

第7章　MLプログラミング入門

SML#言語は，Standard ML 言語と後方互換性のあるプログラミング言語です．SML#の高度な機能を使いこなすために，本章でまず，Standard ML プログラミングの基礎を簡単に解説します．

なお本章の内容は，概ね以下の Standard ML の教科書

大堀 淳．**プログラミング言語** *Standard ML* **入門**．共立出版, 2000.

に準拠しています．

7.1　ML 言語について

SML#は ML 系関数型プログラミング言語の一つです．

ML 言語は Edinburgh LCF[3] のメタ言語（**M**eta **L**anguage）として開発されました．メタという接頭語のこの特別な使われ方は，ギリシャ語の “ta meta ta phusika” という用例に遡るといわれています．このフレーズは単に，自然学（physics）のあとに置かれた名前が付けられていない本を示すギリシャ語でしたが，その本の内容が哲学（形而上学，metaphysics）であったことから，このメタという接頭語に，通常の思考の次元をこえて物事を分析する哲学的な態度，という意味が与えられるようになりました．言語学の文脈では，メタ言語とは，ある言語の分析や記述をするための言語を意味します．プログラミング言語を変換したり処理するための言語もメタ言語とみなすことができます．Edinburgh LCF は計算可能な関数（つまりプログラム）を対象とした一種の定理証明システムであり，その対象となる関数は PPLAMBDA と呼ばれる型付ラムダ計算で表現されました．LCF ML は PPLAMBDA のメタ言語，つまり PPLAMBDA で書かれたプログラムを操作しするためのプログラミング言語でした．ML の名前はこの歴史的事実に由来しています．関数を表すラムダ式を柔軟に操作する目的のために，ML 自身ラムダ計算を基礎とする関数型言語として設計されました．さらにプログラムを操作するプログラムを柔軟にかつ信頼性を持って記述するために，型の整合性を自動的に検査する型推論機構が初めて開発されました．

この LCF ML は，PPLAMBDA の操作言語にとどまらない汎用のプログラミング言語として価値があることが認識され，Cardelli によって LCF とは独立したプログラミング言語としての ML コンパイラが開始され，PDP11 や VAX VMS などの上に実装され使用されました．その後，Milner，Harper，MacQueen 等によってプログラミング言語としての仕様策定の努力がなされ，さらにこの言語が Edinburgh 大学によって Cardelli の ML 上に実装されました．これらの成果を踏まえ，Standard ML の仕様 [5] が確定し，その後改訂され現在の Standard ML 言語仕様 [6] として完成しました．

7.2　宣言的プログラミング

時折，関数型プログラミングは「宣言的」であると言われます．この言葉は厳密な技術的用語ではなく，プログラムが実現したいことそのものの記述するといった意味の曖昧性のある言葉です．しかし，これから書こうとするプログラムが表現すべきものがあらかじめ理解されている場合，宣言的な記述は，より明確な意味を持ちます．

書こうとするプログラムが入力を受け取り出力を返す関数と理解できる場合，そのプログラムを，入力と出力の関係を表す関数として直接表現できれば，より宣言的記述となり，したがってより分かりやすく簡潔なプログラムとなるはずです．ML は，この意味でより宣言的な記述を可能にするプログラミング言

語といえます．ML プログラミングをマスターする鍵の一つは，この意味での宣言的プログラミングの考え方を身につけることです．

簡単な例として，数学の教科書で学んだ階乗を計算するプログラムを考えてみましょう．自然数 n の階乗 $n!$ は，以下のように定義されます．

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

ML では，このプログラムを以下のように記述できます．

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

この宣言によって `fact` と言う名前の関数が定義されます．この定義は “|” によって区切られた 2 つの場合からなっています．上段は，引数が `0` の場合は `1` を返すことを表し，下段はそれ以外の場合，引数 `n` と `n - 1` の階乗とを掛けた結果を返すことを表しています．それぞれが，上記の階乗の定義とほぼ完全に一致していることがおわかりでしょう．

このような単純な関数に限らず，種々のプログラムを，その意図する意味に従い宣言的な関数として表現できれば，複雑なプログラムをより簡潔かつ分かりやすく書き下すことができるかもしれません．プログラミング言語 ML は，この考え方に従い，大規模で複雑な計算をプログラムする言語と言えます．以下に続く節で，その基本となる考え方を学んでいきましょう．

7.3　式の組み合わせによる計算の表現

前節の階乗の計算では，引数が `0` の場合は `1` を返し，それ以外の一般の `n` の場合は，`n * fact (n - 1)` を返すようにプログラムされていました．このように，関数が返すべき値は，値を表す式で表現されます．最初の場合の `1` も，値そのものではなく，ゼロという値を表す式とみなします．`n * fact (n - 1)` は，変数 `n` と関数を含む式です．一般に ML 言語の大原則は，

ML のプログラミングは，必要な値をもつ式を定義することを通じて行う

というものです．

式は以下の要素を組み合わせて構成されます．

1. `0` などの定数式
2. 関数の引数やすでに定義されている値を表す変数
3. 関数呼び出し
4. 関数を含む種々のデータ構造の構成

1 から 3 までの組み合わせは，高校などで慣れ親しんだ算術式と同じ構造を持っています．例えば，1 から n までの自然数の 2 乗の和 $S_n = 1^2 + 2^2 + \cdots + n^2$ は以下の公式で求められます．

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

この式は，ML で直接以下のようにプログラムできます．

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

`*`と `div` はそれぞれ自然数の乗算と除算を表します．この式は n が定義されていれば，正しく S_n の値を計算します．

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

7.4 定数式と組込み関数

型付き言語である ML では，定数式はすべて決まった型を持ちます．ML でよく使われる組込み型には以下のものがあります．

型	内容
int	（マシン語 1 ワードの大きさの）符号付き整数
real	倍精度浮動小数点
char	文字型
string	文字列型
word	（マシン語 1 ワードの大きさの）符号無し整数
bool	真理値型

真理値型以外は，他のプログラミング言語と考え方は同じです．これらの組込み方の定数式と基本組込み演算を覚えましょう．真理値型の扱いは次節で説明します．

7.4.1 int 型

int 型は整数型です．SML#では，C 言語の long と同一の 2 の補数表現で表される 1 語長の符号付き整数です．int 型定数には通常の十進数の記述以外に 0x$\langle h \rangle$ 形式の 16 進数でも記述できます．$\langle h \rangle$ は 0 から 9 の数字と a(A でもよい) から f(F でもよい) までのアルファベットの並びです．負数の定数はチルダ記号~を付けて表します．SML#は int 型の値を十進数で表示します．

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

int 型に対しては，通常の四則演算+，-，*，div が 2 項演算子として定義されています．

```
# 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
```

これらの組合せは，通常の算術演算の習慣に従い，*,div が+，-より強く結合します．また，同一の演算子は左から順に計算されます．

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

7.4.2　real 型

real は浮動小数点データの型です．SML#では，C 言語の double に相当する 64 ビットの倍精度浮動小数点です．real 型の定数は ⟨*s*⟩ . ⟨*d*⟩ E ⟨*s*⟩ の形で表現します．ここで ⟨*s*⟩ は符号付き十進数，⟨*d*⟩ は符号なし十進数です．整数部，小数部，指数部のいずれも省略できますが．小数部と指数部の両方を省略すると real 型ではなく int 型と解釈されます．

```
# 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
```

real 型に対しても int 型同様に四則演算が定義されています．ただし，除算は div ではなく/です．

```
# 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 型

char は 1 文字を表すデータ型です．文字 ⟨*c*⟩ を表す char 型定数は#" ⟨*c*⟩ "と書きます．⟨*c*⟩ には通常文字以外に以下の特殊文字のエスケープが書けます．

\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
```

文字型に対して以下の組込み演算が定義されています．

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

chr は与えられた数字を ASCII コードとみなし，そのコードの文字を返します．ord は文字型データの ASCII コードを返します．

```
# 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 型

string は文字列を表すデータ型です．string 定数は"と"で囲んで文字列表します．この文字列の中に特殊文字を含める場合は，char 型で定義されているエスケープシーケンスを使います．string データには，標準出力にプリントする print と 2 つの文字列を連結する^が組込み関数として定義されています．

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

7.4.5 word 型

word は符号なし整数型です．word 型定数は 0w$\langle d \rangle$ または 0wx$\langle h \rangle$ の形で表現します．ここで $\langle d \rangle$ はそれぞれ十進数，$\langle h \rangle$ は 16 進の数字列です．SML#は word 型定数を 16 進表現で表示します．

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

word 型データに対しても int 型と同じ四則演算が定義されています．さらに word 型データに対しては，種々のビット演算が定義されています．

7.5　bool 型と条件式

条件を含む計算も

ML のプログラミングは，必要な値をもつ式を定義することを通じて行う

という ML の原則に従い式で表されます．式

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

は値を表す式です．その値は，以下のように求められます．

1. 式 $E1$ を評価し値を求める．
2. もしその値が `true` であれば $E2$ を評価しその値を返す．
3. もし $E1$ の値が `false` であれば $E3$ を評価しその値を返す．

この $E1$ の式が持つ値 `true`,`false` はそれぞれ「真」と「偽」を表す型 `bool` の 2 つの定数です．`bool` 型の値を持つ式は，これら定数の他に，数の大小比較や論理演算などがあります．

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

条件式は，これら `bool` 型を持つ式を使って定義します．

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

この条件式も値を持つ式ですから，他の式と自由に組み合わせることができます．

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

7.6　複雑な式と関数

定数，変数，組込み関数の組み合わせだけでは，もちろん，複雑な問題を解くプログラムを記述することはできません．ML プログラミングの原則は式を書くことによってプログラムすることですが，その中で重要な役割を果たす式が関数を表す式です．

関数は，すでに第 7.2 節で学んだように，

fun *funName param* `=` *expr*

の構文で定義されます．この宣言以降，変数 *funName* は，引数 *param* を受け取り，式 *expr* の値を計算する関数として使用可能となります．例えば 1 から n までの自然数の 2 乗の和の公式は，n を受け取る関数とみなせます．

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

この定義は `fun` 構文により直接 ML のプログラムとして定義できます．

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

この関数は，以下のように使用できます．

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

7.7 再帰的な関数

前節で学んだ fun 構文は再帰的な定義を許します．すなわち，この構文で定義される関数 *funName* を，関数定義本体 *expr* の中で使用することができます．このような再帰的な関数定義は，以下のような手順で設計していけば，通常の算術関数と同じように自然に定義できるようになります．

1. 関数 *funName* の振る舞いをあらかじめ想定する．
2. 想定された振る舞いをする関数 *funName* があると仮定し，受け取った引数が *param* の場合に返すべきか値を表す式 *expr* を定義する．

例えば，第 7.2 節で紹介した階乗を計算する関数 fact は以下のように設計できます．

1. fact を階乗を計算する関数と想定します．
2. 階乗を計算する関数 fact を使い，階乗の定義に従い，返すべき値を以下のように定義する．
   (a) 引数が 0 の場合，0 の階乗は 1 であるから，1 を返す．
   (b) 0 以外の一般の引数 n の場合，階乗の定義から，n * fact (n - 1) を返す．

この２つの場合を単に書き下せば，以下の定義が得られます．

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

以上の考え方に従い種々の再起関数を簡単に定義できます．たとえば，すべての数の和を求める関数 sum や引数 n に対して与えられた定数 c の n 乗を計算する power なども，これら関数がすでにあると思うことによって，以下のように簡単に定義できます．

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

sum や power が想定した振る舞いをすると仮定すれば，関数が返す値は正しいので，関数全体の定義も正しいことがわかります．

7.8 複数の引数を取る関数

前節の power 関数はあらじめ定義された定数 C の指数乗を計算する関数でしたが，では，この指数の底 C も引数に加え，n と C を受け取って C の n 乗を計算する関数を定義するにはどうしたらよいでしょうか．これには，２つの書き方があります．以下の対話型セッションでは，その型情報とともに示します．

```
# 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
```

powerUncurry の型情報 `int * int -> int` は，引数を組として受け取り整数を返す関数であることを示しています．これに対して powerCurry の型情報 `int -> int -> int` は，引数を 2 つ順番に受け取り整数を返す関数であることを示しています．上の例から想像される通り，ML では関数定義の引数は，その関数が使われる形で書きます．例えば powerUncurry は `fun powerUncurry (C,n) = ...` と定義されていますから，`powerUncurry (2, 3)` のように使います．

7.9　関数適用の文法

powerUncurry は，複数引数を取る C 言語の関数と同様です．ML 言語の強みは，これ以外に，powerCurry にように引数を順番に受け取る関数が書けることです．これを正確に理解するために，ここで ML の関数式の構文を復習しておきましょう．数学などの式では関数の利用（適用）は，関数の引数を括弧でかこみ $f(x)$ のように記述しますが，ML などラムダ計算の考え方を基礎とした関数型言語では，単に関数と引数を並べて `f x` のように書きます．

式の集合を $expr$ で表すと，定数，変数，関数適用を含む ML の構文は一部は，以下のような文法となります．

$$
\begin{array}{lcll}
expr & ::= & c & \text{（定数）} \\
 & | & x & \text{（変数）} \\
 & | & expr\ expr & \text{（関数適用）} \\
 & | & \cdots & \text{その他の構文}
\end{array}
$$

この構文規則のみでは関数適用式が続いた場合，その適用順番をきめることができません．例えば，式に割り算の構文 $expr/expr$ と足し算 $expr + expr$ を導入した場合を考えてみましょう．この文法は，`60 / 6 / 2 + 1` のような式をゆるしますが，どの割り算が先に行われるのかで 2 通りの解釈が可能です．この算術式では，通常左から括弧があるとみなし，`((60 / 10) / 2) + 1` と解釈されます．このような解釈を，割り算の構文は「左結合」し，かつ足し算の構文より「結合力が強い」，と言います．

ラムダ計算を下敷きとした関数適用をもつ言語では，関数適用式に関して以下の重要な約束があります．

> 関数適用式は左結合し，その結合力は最も強い

関数式は左結合すると約束されているので，`powerCurry 2 3` は `((powerCurry 2) 3)` と解釈されます．一般に，`e1 e2 e3 ⋯ en` と書いた式は `(⋯ ((e1 e2) e3) ⋯ en)` とみなされます．

7.10　高階の関数

`powerCurry 2 3` は `((powerCurry 2) 3)` と解釈されるということは，ML 言語における以下の重要な性質を意味しています．

(powerCurry 2) は関数である.

このように ML では，関数は powerUncurry のように関数定義構文を通じて定義された関数名だけではなくプログラムで作ることができます．このように返された値は，powerCurry の例のように，使うことができます．さらに，名前を付けたら関数に渡したりすることができます．例えば，以下のようなコードを書くことができます.

```
# 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
```

apply に対して推論された型は，この関数が，関数を受け取って整数を返す関数であることを示しています.

以上の例から理解される通り，ML は以下の機能を持っています.

> 関数は，整数型などのデータと同様，プログラムが返すことができる値でああり，自由に受け渡し，使うことができる．特に関数は，別の関数を引数として受け取ることができる.

これが,「プログラムを式で定義していく」という大原則の下での，ML プログラミングの最も重要な原則です．関数を返したり受け取ったりする関数を**高階の関数**と呼びます.

7.11 高階の関数の利用

ML でクールな，つまり読みやすくかつ簡潔なプログラミングを効率良く書いていく鍵は，高階関数の機能を理解し，高階の関数を使い，式を組み合わせていくことです．この技術の習得には，当然習熟が必要でこのチュートリアルで尽くせるものではありませんが，その基本となる考え方を理解することは重要と思います．そこで，以下，簡単な例を用いて，高階の関数の定義と利用に関する考え方を学びましょう.

数学で数列の和に関する Σ 記法を勉強したと思います．この記法は以下のような等式に従います.

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

高校の教科書などでこの記法を学ぶ理由は，もちろん，この Σ で表現される働きに汎用性があり便利だからです．記法 $\sum_{k=1}^{n} f(k)$ の簡単な分析をしてみましょう.

- $\sum$ 記法には k, n, f の 3 つの変数が含まれる．この中で k は，1 から n まで変化することを表す仮の変数である.
- 式 $\sum_{k=1}^{n} f(k)$ は，与えられた自然数 n と関数 f に対して $f(1) + f(2) + \cdots + f(n)$ の値を表す式である.

従って，$\sum$ は n と整数を引数とし整数を返す関数 f を受け取る関数，つまり高階の関数とみなすことができます．高階の関数をプログラミングの基本とする ML では，この関数を以下のように直接定義可能です.

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

この Sigma を使えば，任意の自然数関数の和を求めることができます．

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

高階の関数は，このようにひとまとまりの機能を表現する上で大きな力をもちます．複雑なプログラムを簡潔で読みやすい ML のコードとして書き下していく能力を身につける上で重要なステップは，ひとまとまりの仕事を高階の関数として抜き出し，それらを組み合わせながら，式を作りあげていく技術をマスターすることです．後に第 7.20 節で説明する ML の多相型システムは，このスタイルのプログラムをより容易にする機構といえます．この機構とともに，ML の高階関数を活用すれば，複雑なシステムを宣言的に簡潔に書き下していくことができるようになるはずです．

7.12　ML における手続き的機能

これまで学んできた通り，式の組み合わせでプログラムを構築していく ML 言語はプログラムを宣言的に書く上で最も適した言語の一つです．これまでに，$\sum$ 関数などの算術演算を行う関数を例に説明しましたが，第 7.2 節で説明した通り，宣言的とは，意図する意味をできるだけそのまま表明するということであり，決して数学的な関数としてプログラムを書いていくことが望ましい，との表明ではありません．

もし，実現しようとしている機能が，手続き的な操作によって最も簡潔に表現し理解できるなら，その手続きをそのまま表現するのが望ましいはずです．例えば，みなさんが今ご覧のディスプレイに文字を表示する操作は，これまでに書かれている内容を別な情報で上書きする操作であり，具体的には，ディスプレイのバッファを書き換える手続き的な操作です．ディスプレイに限らず，外部との入出力処理は，外部の状態に依存する本質的に手続き的な操作です．そのほか，例えば新しい名前や識別子の生成などの処理，さらに，サイクルのあるグラフの変換なども，手続き的なモデルで理解しプログラムしたほが，より宣言的な記述になる場合が多いと言えます．

このような操作をプログラムで簡潔で分かりやすく表現するには，手続き的な処理が記述できたほうが便利です．式の組み合わせでコードを書いていくことが基本である関数型言語にこのような手続き的な処理を導入するために必要なことは，以下の２つです．

1. 状態を変更する機能の導入．例にあげたディスプレイの表示内容は，抽象的にはディスプレイの状態と捉えることができます．手続き的なプログラムの基本は，この状態を変更することをくり返し目的とする状態を作りだすことです．そのためには，変更可能なデータが必要となります．

2. 評価順序の確定．ディスプレイの表示内容の変更操作が複数あった場合，それらの適用順序によって，どのような像が見えるかが決まります．従って，手続き的なプログラムを書くためには，変更操作がどの順に行われるかに関する知識と制御が必要となります．

ML には，これら２つが関数型言語の枠組みの中に導入されています．

7.13　変更可能なメモリーセルを表す参照型

ML には，以下の型と関数が組み込まれています．

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

型'a ref は「'a の値への参照」を表す型です．参照とはポインタのことです．関数 ref は，任意の型'a の値を受け取り，その値への参照を作成し返す関数です．関数!は，参照を受け取りそれが指す値を返す関数です．関数:=は代入を行う関数，すなわち，参照と値を受け取り，参照が指す値を受け取った値に変更する関数です．以下は，参照型を使用した対話型セッションの例です．

```
# 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
```

この参照型を，状態を参照として保持することにより状態に依存する処理などの本質的に手続き的な処理を，通常の関数として書くことができます．

7.14 作用順，左から右への評価戦略

手続き的処理を書くためにはプログラムの各部分の実行順序を決める必要があります．値を表す式を組み合わせてプログラムを書いていく関数型言語では，書かれたプログラムの実行は，式の値を求めることに対応します．これを，式の評価と呼びます．ML では，式の評価は以下のような順で行われます．

1. 式は，同一レベルであれば，左から右に評価する．たとえば，(power 2) (power 2 2) の場合，まず (power 2) が評価され 2 乗する関数が得られ，次に (power 2 2) が評価され 4 が得られ，最後に，2 乗する関数に 4 が適用され，14 が得られます．また，宣言の列は，宣言の順に評価されます．
2. 関数の本体は，引数に適用されるまで評価されない．

さらに，手続き的な処理を容易にするために，以下の逐次評価構文が用意されています．

$$
\begin{array}{lcll}
expr & ::= & \cdots & \\
 & | & (expr; \cdots ; expr) & \text{（逐次評価構文）}
\end{array}
$$

$(expr_1; \cdots ; expr_n)$ は，$expr_1$ から $expr_n$ まで順番に評価し，最後の式 $expr_n$ の値を返す構文です．

以上の評価順序のもとで参照型を使えば，手続き的な処理を関数型言語の枠組みで，簡単に書くことができます．たとえば新しい名前を生成する関数は，以下のように書けます．

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

7.15　手続き的制御

手続き的な処理を学んだところで，手続き的な言語での制御構造の記述と式による関数の記述の関係を振り返ってみましょう．状態の変更を伴う手続き的処理は，外部の状態を変更する I/O などの機能の表現には欠かせないものです．これらは，手続き的な操作として表現するのがもっとも分かりやすいプログラムとなります．手続き型言語では，これらに加え，ループや条件式などの種々の制御構造も手続き的な機能で実現されるのに対して，ML では式の組み合わせで表現されます．これまで見てきた通り，ML ではプログラム構造は手続き的機能に関係のない概念ですから，表現すべき内容を式で表す ML の方がより宣言的で理解しやすいプログラムと言えます．

しかし，この点に注意が必要です．C のような手続き型言語では，手続き的処理が基本ですから，階乗を求める関数 fact は，再帰ではなく，ループを使って以下のように書くのが一般的と思われます．

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

このようにループを使って書くほうが，再帰を使った書き方，たとえば ML での

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

のようなコードより効率的です．しかし，この事実は，関数型言語の本質的な非効率さを意味するものではありません．

7.16　ループと末尾再帰関数

くり返し処理（ループ）の設計の基本は，

1. 求める値に至る（それを特別な場合として含む）計算の途中状態を設計し、
2. 計算（の途中）結果を保持するための変数と
3. 計算の進行状態を保持する変数，

を用意し最終状態のチェックしながら，これら変数を更新するコードを書くことです．1 から n までの積を求める場合は，

- 途中の計算状態：$s = i*(i+1)*...*n$
- 計算結果を持つ変数：s
- 計算の状態を保持する変数：i

と考え，最終状態（$i = 0$）をチェックし変数を変更するコードを書くと前の fact のようなコードとなります．この考え方に従えば，種々の複雑な処理をループとして書き下すことができます．

この考え方は，手続き型言語に特有のものではありません．手続き型言語では，計算結果を持つ変数と状態を制御する変数の値を変更しながらループします．この変数の変更とループは，

変数の現在の値を受け取り，新しい値を生成する関数

と考えると，自分自身を呼び出すだけの関数となります．C 言語の fact 関数で表現されたループコードは，以下の ML コードで実現できます．

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

この loop が行う自分自身を呼び出すだけの再帰関数を末尾再帰と呼びます．末尾再帰は，手続き型言語のループと同様の実行列にコンパイルされ効率よく実行されます．

7.17 let 式

ML プログラミングの基本は，種々の式を定義しそれらに名前を付け組み合わせていくことです．これまで見てきた例では，名前はすべてトップレベルに記録されていました．しかし，大きなプログラムでは，一時的に使用する多数の名前が必要となり，それら名前を管理する必要があります．例えば fact は，トップレベルに定義された loop を使って定義されていますが，この名前は fact の定義のためだけに導入されたものであり，他の関数ではべつなループ関数が必要です．そこで，これら特定の処理にのみ必要な名前のスコープ（有効範囲）を制限できれば，より整理され構造化されたコードとなります．そのために，以下の let 構文が用意されています．

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

let と in の間には，in と end の式の中だけで有効な宣言が書けます．例えば，末尾再帰の fact 関数は，通常以下のように定義します．

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

7.18 リストデータ型

ML プログラミングでよく使用される複合データはリストと高階多相関数です．リストを用いたプログラムには，ML プログラミングの基本である以下の要素が含まれています．

- 再帰的データの生成と処理
- パターンマッチングによるデータ構造の分解
- 多相型高階関数

リストを用いてこれら機能をマスターしていきましょう.

リストは要素の並びです. ML では “[” と “]” の間に要素の式を並べると，リストが定義されます.

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

この式は，以下の式の略記法です.

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

この構文とその評価結果を理解しましょう.

- e :: L は要素を表す式 e とリストを表す式 L から，リスト L の先頭に e を付け加えて得られるリストを生成する式.
- nil は空のリストを表す定数.
- int list は int 型を要素とするリストを表す型

7.19　式の組み合わせの原則

ML プログラミングの原則は，式を組み合わせていくことでしたが，もちろんどのような組み合わせでも許されるわけではありません. ML では，プログラムを構成する際，以下の基本原則に従います.

式は型が正しい限り自由に組み合わせることができる

リストの場合を例にこの原則を考えてみましょう. リストは，その要素の型に制限はありません. どのような値であれ，同じものは同一のリストにすることができます. 以下の対話型セッションは，様々な型のリストを構築しています.

```
# fact 4 :: 4 + 4 :: (if factorial 1 = 0 then nil else [1,2,3]);
val it = [24, 8, 1, 2, 3] : 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 と explode はそれぞれ文字のリストを文字列に変換する関数および文字列を文字のリストに変換する関数です.

7.20 多相型を持つ関数

「式は型が正しい限り自由に組み合わせることができる」という原則を最大限に活かすためには，組み合わせを実現する関数は，種々の型に対応している必要があります．前節のリストを構成する関数（演算子）::の型は以下の通りです．

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

opは，中置演算子を関数適用の構文で使うための接頭語です．この式は，::が，任意の型'aとそれと同じ型'aの要素とするリスト'a listを受け取って，'aの要素とするリスト'a listを返す関数を表しています．ここで使われる'aは任意の型を表す型変数です．また['a.···]は「すべての型'aについて，···」である，との表明を表します．このように型変数を含み，種々の型のデータに利用できる関数を**多相関数**と呼びます．さらに，これら多相関数を組み合わせて定義された関数も以下のように多相型を持ちます．

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

MLは，式の組み合わせによるプログラミングを最大限にサポートするために，関数定義に対して，その関数のもっとも一般的な使い方を許すような，**最も一般的な多相型**を推論します．以下の関数定義を考えてみましょう．

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

この関数は，関数と引数を受け取り，関数を引数に２回適用します．このふるまいを考えると，以下の型が最も一般的な利用を許す型であるとが理解できるでしょう．

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

実際MLでは以下のように推論されます．

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

第8章　SML#の拡張機能：レコード多相性

本章以降の各章では，SML#で導入された Standard ML の拡張機能を例を用いて説明します．

まず本章では，レコード多相性を基礎とした ML でのレコードを用いたプログラミングを学びます．レコード多相性は，特別な付加機能ではなく，ML の原則「式は型が正しい限り自由に組み合わせることができる」に従ってレコードを含むプログラミングを行う上での基本機能です．Standard ML にはこの機能が欠けているため，レコードの特性を生かした ML スタイルのプログラムが書けませんでした．この理由から，本書では，レコードの説明を遅延していました．まず，レコードの基礎から学びましょう．

8.1　レコード構文

ML ではレコード式は以下の文法で定義します．

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

l はラベルと呼ぶ文字列です．レコードの定義の簡単な例を示します．

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

1 から始まる連続した数字をラベルとして持つレコードは，組み型と解釈され組として表示されます．

```
# {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
```

以前第 7.8 節で説明した複数引数の関数は `powerUncurry (n, C)` のように定義しましたが，これも数字をラベルとしたレコードの受け取ることを表しています．

「式は型が正しい限り自由に組み合わせることができる」という ML の原則に従い，レコードの要素には ML で定義できる任意の型を含むことができます．従って，レコードを生成する関数に対しても，リストの場合と同様に多相型が推論されます．

```
# 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　フィールド取り出し演算

レコードに対する演算は，レコードのフィールドの取り出しです．このために，組み込み関数

#l

が用意されています．この関数はラベル l を含むレコードを受け取り，そのラベルのフィールドの値を返す関数です．ML の多相型の原則に従えば，この関数は，ラベル l を含む任意のレコード型からそのフィールドの値の型への関数です．SML#では，以下のような型が推論されます．

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

表現'a#{X:'b} は，型が'b のフィールド X を含む任意のレコード型を代表する型変数です．従って，推論された型は，X を含む任意のレコードを受け取り，X のフィールドの値を返す関数を表しており，#X の動作を正確に表現しています．この関数の適用例を以下に示します．

```
# #X {X = 1.1, Y = 2.2}
val it = 1.1 : real
```

これら演算を含む関数に対しては，レコードに関する多相型が推論されます．

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

この関数は，引数に対して X と Y のフィールド取り出しが行われているため，引数は少なくともこれら 2 つのフィールドを含むレコードでなければなりません．SML#が推論する型は，その性質を正確に反映した最も一般的な多相型です．

8.3 レコードパターン

レコードのフィールド取り出しは，パターンマッチングの機能を用いても行うことができます．Standard ML には以下のパターンが含まれています．

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

最初のパターンは指定されたフィールドからなるレコードにマッチするパターン，２番目のパターンは少なくとも指定されたフィールドを含むレコードにマッチするパターンです．フィールドはフィールド名のみ書くこともできます．その場合，フィールド名と同じ変数が指定されたものとみなされます．以下は，レコードパターンを用いたフィールド取り出しの例です．

```
# 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]
```

このレコードパターンは，他のパターンと自由に組み合わせて使用できます．

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

この例では，X フィールドを含むレコードのリストの先頭のレコードの X フィールドの値を返しています．

8.4 フィールドの変更

SML#言語では，以下の構文によるレコードのフィールドの（関数的）変更が定義されています．

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

この構文は，レコード式 $expr$ の値をそれぞれ指定された値に変更して得られるレコードを生成します．この構文は，常に新しいレコードを生成し，もとのレコードは変更されません．この構文も ML の型の原則に従いレコード多相性を持ちます．以下は，この構文含む関数の例とその型推論の例です．

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

この構文とレコードパターンを組み合わせた以下の形の関数は，レコードを扱うプログラムでよく登場するイディオムと言えます．

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

関数 reStructure は，従業員レコード p を受け取り，その Salary フィールドを 8.03%減額する関数です．型情報から理解されるとおり，この関数は，Salary フィールドを含む任意の従業員レコードに適用可能な，汎用の減額関数となっています．

8.5 レコードプログラミング例

多相レコード演算機能をもった言語（現在のところ SML#しかありませんが）では，レコード構造は，種々の属性に着目したモジュール性の高いプログラムを型安全に構築していく上で大きな武器となります．これによって，多相関数による汎用性あるプログラムが，大規模なプログラム開発にスケールするようになります．

そのような例として，放物線を描いて落下する物体をプロットする場合を考えてみましょう．物体の位置は直交座標で表現するとします．そして，物体は，現在位置を表す X:real と Y:real のフィールドと現在の速度を表すを含むレコード Vx:real と Vy:real のフィールドを含むレコードで表現することにします．これだけを決めておけば，物体のその他の属性や構造とは独立に，放物線上を移動する関数を設計できます．物体の運動は，物体を単位時間経過後の位置に移動させる関数 tic を書き，この関数をくり返し呼び続ければ実現できます．

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

簡単のために単位時間を 1 とします．直行座標系での物体の移動ベクトルは，それぞれの座標のベクトルの合成ですから，X 軸と Y 軸について独立に関数を書きそれを合成すればよいはずです．位置は，それぞれ現在の位置に，単位時間あたりの移動距離，つまり速度を加えればよいので，X 軸方向は Y 軸方向それぞれ，独立に以下のようにコードできます．(対話型セッションで型と共に示します．)

```
# 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]
```

次に，それぞれの軸方向の速度を変更する関数を書きます．X 軸方向は等速運動をし，Y 軸方向は重力加速度による等加速度運動をする場合の関数は，位置の変更と同様，以下のように簡単にコードできます．

```
# 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 は変化がないので，不要ですが，雛形として記述しておくと将来の変更に便利です．

単位時間後の物体を求める関数 tic は，以上の関数を合成することによって得られます．

```
fun tic (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
```

このようにして得られたプログラムはモジュール性に富みかつ型安全で堅牢なシステムとなります．たとえば，水平方向の速度は風の抵抗に現在の速度に対して 10%減少していく運動なども，accelerateY を

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

と変更するだけで実現できます．

この tic 関数に対しては，本節の冒頭で書いた型が推論され，位置と速度属性をもつ任意の物体に適用可能です．

8.6　オブジェクトの表現

レコードは種々のデータ構造の定義の基本であり，データベースやオブジェクト指向プログラミングなどで使われています．第 12 章で詳しく説明する通り，多相レコード操作を基本に，データベースの問い合わせ言語 SQL を完全な形で ML 言語内にシームレスに取り込むことができます．オブジェクト指向プログラミングに関しては，計算モデルが異なるため，その機能を完全に表現することはできませんが，オブジェクトの操作に関しては，多相型レコードで表現できます．

オブジェクトは，状態を持ち，メソッドセレクタをメッセージとして受け取り，オブジェクトの属するクラスの対応するメソッドを起動し状態を更新します．クラスは，メッセージによって選び出されるメソッド集合ですから，関数のレコードで表現できます．各メソッド関数は，オブジェクト状態を受け取りそれを更新するコードです．例えば，X 座標，Y 座標，Color 属性を持ち得る pointClass のオブジェクト表現を考えてみましょう．オブジェクトは，例えば{X = 1.1, Y = 2.2}のようなレコードへの参照を内部に持つとします．すると，各メソッドはこのオブジェクトを self として受け取り更新する関数と表現できます．例えば X 座標に新しい値をセットするメソッドは，以下のようにコードできます．

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

このメソッドは，X を含む任意のオブジェクトに適用できます．これらメソッドにメソッド名を付けてレコードにしたものをクラスと考えます．例えば，pointClass は以下のように定義できます．

```
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})
    }
```

オブジェクトは，メッセージを受け取り，このクラスの中からメソッドスイートの中から，対応する関数を選択し自分の状態に適用する関数です．関数を値として使用できる ML では，以下のようにコードできます．

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

Color 属性をもつオブジェクトも同様です．

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

この定義の下で，以下のようなオブジェクト指向スタイルのコーディングが可能です．

```
# 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
  ...
```

最後の例のように，完全な静的な型チェックも保証されています．

8.7 多相バリアントの表現

レコード構造は，各要素に名前を付けたデータ構造です．この構造の双対をなす概念にラベル付きバリアントがあります．多くの方にとって不慣れな概念と思われますが，必要とされる機能はレコードとほぼ

同一であるため，多相レコードが表現できれば，多相バリアントも表現できます．多相バリアントに興味のある方への参考に，以下，その考え方とコード例を簡単に紹介します．

レコードが名前付きのデータの集合であるのに対して，ラベル付きバリアントは，名前付きの処理の集合の下での処理要求ラベルのついたデータと考えることができます．直行座標系と極座標系の両方のデータを扱いたい場合を考えます．それぞれのデータは処理の仕方がちがいますから，データにどちらの座標系かを表すタグ（名前）を付けます．これらデータを多相関数と一緒に使用できるようにする機構が多相バリアントです．これは，バリアントタグを，メソッド集合から適当なメソッドを選び出すセレクタと考えると，多相レコードを使って表現できます．同一の点の２つのそれぞれの座標系での表現は，例えば以下のようにコード可能です．

```
# 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]
```

つまりタグ T をもつバリアントは，タグ T を処理するメソッドスイートを受け取り，自分自身にその処理を呼び出す行うオブジェクトと表現します．あとは，必要な処理をそれぞれの表現に応じて書き，タグに応じた名前をもつレコードにすればよいわけです．例えば，原点からの距離を計算するメソッドは以下のように実現できます．

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

多相バリアントデータをメソッドスイートに適用することによって起動できます．

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

これによって，表現の異なるデータのリストなどを多相関数によって安全に処理できるようになります．

第９章　SML#の拡張機能：その他の型の拡張

SML#では，レコード多相性に加え，以下の２つの拡張をしています．

1. ランク１多相性
2. 第一級のオーバーローディング

本章では，これら機能を簡単に説明します．

9.1　ランク１多相性

Standard ML の多相型システムが推論できる多相型は，すべての多相型変数が最も外側で束縛される型です．例えば関数

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

に対しては，

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

の型が推論されます．SML#では，このような関数に対して以下のようなネストした多相型を推論可能に拡張しています．

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

この型は，'a としてある型τを受け取って多相関数 ['b. 'b -> τ * 'b] を返す関数型です．実際，以下のような動作をします．

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

型変数をt，種々の型にインスタンス化ができない単相型をτ，多相型をσで表すと，Standard ML 言語の多相型はおおよそ以下のように定義されるランク０と呼ばれる型です．

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

これに対して，SML#で推論可能な多相型は，多相型が関数の引数の位置以外の位置にくることを許す以下のようなランク１と呼ばれる型です．

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

この拡張は，もともとレコード多相性の効率的な実現を目指した技術的な拡張であり，純粋な ML の型理論の枠組みでは殆ど違いはありません．しかし，Standard ML の改訂版で導入された値多相性制約のもとでは，重要な拡張となっています．次節で値多相性とランク１多相性の関連を解説します．

9.2　ランク 1 多相性による値多相性制約の緩和

ML の多相型システムは，第 7.13 節で学んだ手続き的機能を導入すると，整合性が崩れることが知られています．例えば参照型構成子 ref は，任意に型の参照を作ることができるので，ML の型システムの原則に従いナイーブに導入すると，以下のような多相型を持つように思います．

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

しかし，この型付けのもとで ML の型推論を実行すると，以下のようなコードが書けてしまい，型システムが破壊されていまいます．

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

まず 1 行目で idref の型が ['a.?('a -> 'a) ref] と推理されます．この型は (int -> int) ref としても使用できる型です．また，代入演算子:=も多相型を持ちますから (int -> int) ref 型と int -> int 型を受け取ることができます．従って 2 行目の型が正しく受け付けられ実行されます．この時 idref の値は int -> int の値への参照に変更されていますが，その型は以前のまま ['a. ('a -> 'a) ref] です．従って 3 行目の型が正しく，実行されてしまいますが，その結果は，算術演算が文字列に適用され実行時型エラーとなります．

この問題を避けるために，ML では，

> 多相型を持ち得る式は値式に限る

という値多相性制約が導入されています．値式とは，計算が実行されない式のことであり，定数や変数，関数式 fn x => x，それらの組などです．関数呼び出しを行う式は値式ではありません．たとえば，ref (fn x => x) はプリミティブ ref が呼び出されますから値式ではありません．この制約によって，関数呼び出しの結果に多相型を与えることが禁止されます．これによって，参照を作り内部に参照を含む ref (fn x => x) のような式は多相型を持てなくなり，上記の不整合が避けられます．

SML#もこの制約に従っています．しかし，SML#では，多相型が組の要素や関数の結果などの部分式にも与えることができるため，この値式制約が限定されたものとなり，ユーザにとって大幅にその制約が緩和される効果があります．例えば以下の関数定義と関数適用をみてみましょう．

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

Standard ML の型システムでは，関数 f には

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

のような多相型があたえられますが，この関数の適用 f 1 の結果に現れる型変数を再び束縛することは値多相性制約に反しますので，g には多相型を与えることができません．これに対して SML#では，ランク 1 多相性により，この関数に以下の型が推論されます．

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

この結果の型に対して推論された多相型は，関数適用の前に推論されたものであり，関数適用の結果に対して推論されたものではありません．従って，関数適用の結果はその多相型に'a の実際の値が代入された以下のような多相型となります．

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

9.3 第一級オーバーローディング

ML でよく使われる組込み演算は，オーバーロードされています．例えば，加算演算子+は，int，real，word などを含む型の加算に使用できます．

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

この+は多相関数と違い，適用される値によって対応する組込み演算（上の例の場合 Int.+，Real.+，Word.+ のどれか）が選択されます．Standard ML では，このオーバーロード機構はトップレベルで解決する戦略がとられています．トップレベルでもオーバーロードが解消しなかった場合，コンパイラはあらかじめ決められた型を選択します．例えば，関数

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

を定義すると，演算子のオーバーロードを解決するための x に関する型情報がないため，あらかじめ決められている int 型が選択されます．

この戦略は，データベースの問い合わせ言語をシームレスに取り込む上で，大きな制約となります．SQL で提供されている種々の組込み演算の多くは，データベースのカラムに格納できる型に対してオーバーロードされています．これらの演算子の型をすべて SQL 式が書かれた時点で決定してしまうと，SQL の柔軟性が失われ，SML#からデータベースを柔軟に使用する上での利点を十分に活かすことができません．そこで SML#では，オーバーロードされた演算子を，第一級の関数と同様に使用できる機構を導入しています．オーバーロードされた演算子を含む式は，その演算子が対応可能な型に関して多相的になります．例えば関数 plus に対して以下のような型が推論されます．

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

この関数は，型変数'a に int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32（int16 以下は省略されています）の何れかを代入しえ得られる型をもつ関数として使用できます．型変数に付加されている制約::{...}は，オーバーロード演算子に許される型に制限があるための制約です．

第10章　SML#の拡張機能：Cとの直接連携

SML#は，C 言語とのシームレスな直接連携をサポートしています．C 言語は事実上のシステム記述言語です．OS が提供するシステムサービスなども，C 言語のインターフェイスとして提供されています．SML#では，これら機能を，特別なライブラリなどを開発することなく直接利用することができます．本節では，その利用方法を学びます．

10.1　C 関数の使用の宣言

SML#から C 言語で書かれた関数を呼び出すために必要なコードは，その関数の宣言のみです．宣言は以下の文法で記述します．

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

symbol は使用する C 関数の名前，*type* はその型の記述です．型については次節で説明します．この宣言によって，SML#コンパイラは，C 言語でコンパイルされたライブラリやオブジェクトファイルの中の指定された名前を持つ関数をリンクし，SML#の変数 *id* として利用可能にします．リンク先のコードは，SML#が動作している OS のシステム標準の呼び出し規約に従っているものであれば，C 言語以外でコンパイルされたものでも構いません．このリンクはコンパイル時に行われます．従って，この`_import`宣言を含む SML#プログラムをリンクするときは，*symbol* をエクスポートするライブラリまたはオブジェクトファイルを SML#プログラムと共にリンクする必要があります．ただし，C の標準ライブラリの一部（Unix 系 OS では libc，libm）や pthread ライブラリに含まれる関数は，それらライブラリは SML#コンパイラが常にリンクするため，リンクの指定無しに使用することができます．

この宣言は，`val`宣言を書けるところならどこに書くことができます．この宣言で定義された SML#の変数 *id* は，SML#で定義された関数と同様に使用することができます．

例えば，C の標準ライブラリでは，以下の関数が提供されています．

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

この関数は，文字列を受け取り，その文字列と改行文字を標準出力に印字し終了状態を返します．終了状態は，正常なら印字した文字数，正常に印字できなかった場合は整数定数`EOF`（具体的な値は処理系依存．Linux では -1）です．この型は SML#では，`string -> int`型に対応します．この関数を使用したい場合は，以下のように宣言します．

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

このように，`_import`キーワードの後に C 言語関数の名前と型を宣言するだけで，通常の ML 関数と同様に使用することができます．以下は，対話型セッションでの宣言と利用の例です．

```
# 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","2nd","2012."];
```

```
I
became
fully
operational
in
April
2nd
2012.
val it = [2, 7, 6, 12, 3, 6, 4, 5] : int list
```

この例から理解されるとおり，インポートされた C 関数も，ML プログラミングの原則「式は型が正しい限り自由に組み合わせることができる」に従い，SML#のプログラムで利用できます．

10.2　C 関数の型

SML#から利用可能な C 関数は，SML#の型システムで表現できる型を持つ関数です．_import 宣言には，インポートする C 関数の名前に加え，その C 関数の型を ML ライクな記法で記述します．val 宣言で与えられた変数には C 関数が束縛されます．この変数の型は，C 関数の型から生成された SML#の関数型です．以下，_import 宣言に書く C の関数型の構文と，それに対応する ML の型のサマリを示します．

_import 宣言には以下の形で C 言語の関数の型を書きます．

$(\tau_1, \tau_2, \cdots, \tau_n)$ -> τ

これは n 個の $\tau_1, \tau_2, \ldots, \tau_n$ 型の引数を受け取り，τ 型の値を返す関数を表します．引数が 1 つの場合は引数リストの括弧を省略できます．引数と返り値の型に何が書けるかは後述します．関数が引数を何も受け取らない場合，引数リストの括弧だけを書きます．

() -> τ

関数が返り値を返さない場合（返り値の型が void 型の場合），返り値の型として () を指定します．

$(\tau_1, \tau_2, \cdots, \tau_n)$ -> ()

C 関数が可変長の引数リストを持つ場合は以下の記法を使います．

$(\tau_1, \cdots, \tau_m, \ldots(\tau_{m+1}, \cdots, \tau_n))$ -> τ

これは，m 個の固定の引数とそれに続く 0 個以上の可変個の引数を取る関数の型です．SML#は可変個の関数引数をサポートしていないため，可変部分に与える引数のリストも指定する必要があります．この記法は，可変引数リストに $\tau_{m+1}, \ldots, \tau_n$ 型の引数を与えることを表します．

C 関数の引数および返り値の型には，C の型に対応する SML#の型を指定します．以下，C 関数の引数および返り値に書ける SML#の型を，相互運用型と呼びます．相互運用型には以下の型を含みます．

- IntInf を除く全ての整数型．int，word，char など．
- 全ての浮動小数点数型．real，Real32.real など．
- 要素の型が全て相互運用型であるような組型．int * real など．
- 要素の型が相互運用型であるようなベクタ型，配列型，および ref 型．string，Word8Array.array，int ref など．

これら相互運用型と C の型の対応はおおよそ以下の通りです．

- 整数型および浮動小数点数型の対応は以下の通りです.

SML#の相互運用型	対応する C の型	備考
`char`	`char`	C の `char` 同様，符号は処理系に依存します
`word8`	`unsigned char`	1 バイトは 8 ビットであることを仮定します
`int`	`int`	処理系定義の自然な大きさの整数
`word`	`unsigned int`	
`Real32.real`	`float`	IEEE754 形式の 32 ビット浮動小数点数を仮定します
`real`	`double`	IEEE754 形式の 64 ビット浮動小数点数を仮定します

- τ `vector` 型および τ `array` 型は，τ 型に対応する C の型を要素型とする配列へのポインタ型に対応します．`array` 型のポインタが指す先の配列は C から書き換えることができます．一方，`vector` 型のポインタが指す配列は書き換えることはできません．すなわち，`vector` 型に対応する C のポインタ型には，ポインタが指す先のデータの型に `const` 修飾子が付きます.

- `string` 型は `const char` 型を要素型とする配列へのポインタ型に対応します．このポインタが指す先の配列の最後には必ずヌル文字が入っています．従って，SML#の `string` 型の値を C のヌル終端文字列へのポインタとして使用することができます.

- τ `ref` 型は，τ 型に対応する C の型へのポインタ型に対応します．ポインタが指す先には書き換え可能な要素数 1 の配列があります.

- 組型 τ_1 `*` $\cdots$ `*` τ_n は，$\tau_1, \ldots, \tau_n$ に対応する型のメンバーをこの順番で持つ構造体へのポインタ型に対応します．このポインタが指す先の構造体は C から書き換えることはできません．$\tau_1, \ldots, \tau_n$ が全て同じ型ならば，その型を要素型とする配列へのポインタにも対応します.

C 関数には，SML#の値がそのままの形で渡されます．C 関数の呼び出しの際にデータ変換が行われることはありません．従って，SML#で作った配列を C 関数に渡し，C 関数で変更し，その変更を SML#側で取り出すことができます.

C 関数全体の型は，引数リストを組型とする ML の関数型に対応付けられます．ただし，C 関数の引数や返り値に書ける相互運用型には，以下の制約があります.

- 配列型や組型など，C のポインタ型に対応する相互運用型を，C 関数の返り値の型として指定することはできません.

インポートできる C 関数の型や，SML#と C の型の関係の詳細は，第 III 部の第 29 章を参照ください.

10.3 基本的な C 関数のインポート例

いくつかの C の標準ライブラリ関数をインポートしてみましょう．まず，インポートする C 関数のプロトタイプ宣言を探します．ここでは以下の C 関数をインポートすることにします.

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

次に，引数および返り値の型に対応する SML#の相互運用型を用いて，SML#プログラム中に `_import` 宣言を書きます.

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

これらの C 関数は以下の型の関数として SML#にインポートされます.

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

可変長引数リストの記法を用いれば，printf(3) 関数をインポートすることも可能です．printf 関数のプロトタイプ宣言は以下の通りです.

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

このプロトタイプ宣言に対応付けて，以下の_import 宣言で printf 関数をインポートします.

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

この printfIntReal 関数の型は以下の通りです.

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

ただし，このようにしてインポートした printfIntReal 関数を呼び出す際は，その第一引数は，追加の引数として整数と浮動小数点数をこの順番で要求するような出力フォーマットである必要があります.

ポインタ型と対応する相互運用型の取り扱いには注意が必要です．C 言語ではポインタ引数を，巨大なデータ構造を参照渡しする場合と，関数の結果をストアするためのメモリ領域を指す場合の両方に用います．ポインタ引数の使い方の違いによって，対応する相互運用型は異なります．従って，ポインタ型の引数を持つ C 関数をインポートする場合は，そのプロトタイプ宣言を見るだけでなく，そのポインタ引数の意味をマニュアル等で調べ，その使われ方に正しく対応する相互運用型を選択する必要があります.

ポインタを引数に取る C 関数をいくつかインポートしてみましょう．例えば，C 標準ライブラリ関数 modf をインポートすることを考えます．この関数のプロトタイプ宣言は以下の通りです.

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

この関数の場合，第二引数のポインタは，計算結果の出力先を表します．従って，この関数をインポートするときは，書き換え可能な値を持つ相互運用型を第二引数の型として指定します.

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

char へのポインタ型には特に注意が必要です．C 言語ではこの型を，ヌル終端文字列を表す型として使ったり，最も汎用的なバッファの型として使ったりします．例えば，C 標準ライブラリ関数 sprintf を考えます．そのプロトタイプ宣言は以下の通りです.

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

2 つ現れる char へのポインタは，第一引数は出力先バッファ，第二引数はヌル終端文字列を表します．このポインタの使われ方の違いに従って，これら 2 つのポインタ型に異なる相互運用型を当てはめ，以下のようにインポートします.

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

ここまでは，C 標準ライブラリ関数を例に，C 関数をインポートする方法を説明してきました．しかし，SML#は，ユーザーが書いた C 関数を含む，任意の C 関数をインポートすることができます．SML#からユーザー定義の C 関数に構造体の受け渡す簡単な例を図 10.3 に示します.

sample.c ファイル：

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

sample.sml ファイル:

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

実行例:

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

図 10.1: 構造体引数を持つユーザー定義 C 関数の呼び出し例

10.4 動的リンクライブラリの使用

これまで説明した C 関数宣言

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

は *symbol* の名前を持つ C 関数を静的にリンクする指示です．SML#は，この宣言を含むプログラムをオブジェクトファイルにコンパイルする時，C 関数の名前をリンカによって解決すべき外部名として書き出します．C 関数はリンク時に SML#のオブジェクトファイルと共にリンクされます．対話型モードも第 14 章で説明する分割コンパイルのしくみを使って実装されているため，この宣言は対話型モードでも使用できます．

しかしながら，実行時にしか分からないライブラリの関数などを利用したい場合などは，この静的なリンクは使用できません．SML#は，そのような場合も対応できる動的リンク機能を以下のモジュールとして提供しています．

```
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
```

これら関数は，Unix 系 OS で提供されている同名のシステムサービスと同等の機能をもっています．

samle.c ファイル：

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

実行例:

```
$ 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
```

図 10.2: C 関数の動的リンク

- dlopen は，共有ライブラリの名前を受け取り，共有ライブラリをオープンします.
- dlopen' は，より詳細なオープン機能を指定できます．scope と mode は，OS のシステムサービス dlopen にそれぞれ RTLD_LOCAL，RTLD_GLOBAL および RTLD_LAZY, RTLD_NOW を指定します．詳しくは，OS の dlopen 関数のマニュアルを参照ください.
- dlsym は， dlopen でオープンされた共有ライブラリとライブラリ内の関数名を受け取り，その関数へのポインタを返します.
- dlclose は，共有ライブラリをクローズします.

dlsym で返される関数ポインタは，以下の構文によって SML#の関数に変換することができます.

exp : _import *type*

exp は，codeptr 型を持つ SML#の式です．*type* は，静的リンクを行う _import 宣言で記述する型と同一の構文によって記述された C の型です．この式は，対応する SML#の型を持つ式となります.

動的リンクライブラリの利用手順は以下の通りです.

1. C 言語などで動的リンクライブラリを作成します．例えば Linux で gcc コンパイラを用いる場合，-shared スイッチを指定すれば作成できます.
2. SML#で以下のコードを実行します.
   (a) dlopen でライブラリをオープンします.
   (b) dlsym で関数ポインタを取り出します.
   (c) 型を指定し，C 関数を SML#の変数に束縛します.

図 10.2 に動的リンクの利用例を示します．共有ライブラリの作成は SML#によるコードの実行前であればいつでもよいので，リンク時には存在しない C 関数なども，利用することができます.

第11章　SML#の拡張機能：マルチスレッドプログラミング

SML#は，前章で述べたC言語とのシームレスな直接連携機能を通じて，マルチコアCPU上で並行に動くスレッドを直接サポートします．SML#は2つのマルチスレッドプログラミング方式を提供します．ひとつは，OSが提供するPOSIXスレッド（Pthread）ライブラリを直接使用する方式です．もうひとつは，軽量細粒度スレッドライブラリMassiveThreadsを利用する方式です．いずれの方式においても，SML#で書いたルーチンを，複数のスレッドで，OSやMassiveThreadsが提供するスケジューリング機構をそのまま利用して並行に動かすことが可能です．本章では，SML#を用いたマルチスレッドプログラミングの概要を学びます．

11.1　Pthreadsプログラミング

OSが提供するPthreadsライブラリをほぼそのままバインドした`Pthread`ストラクチャが標準で提供されています．スレッドを生成する`pthread_create`関数および結合する`pthread_join`関数は以下の名前と型で提供されます．

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

マルチスレッドプログラミングの例として，これらの関数を組み合わせ，時間のかかる計算を別スレッドで行うプログラムを書いてみましょう．以下は，`fib 42`をバックグラウンドで計算するプログラムを，対話モードで書いた例です．

```
# 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
```

SML#は独自のスレッドプリミティブを用意していません．また，SML#は，Pthreadライブラリに対して，何か特別なことをしているわけでもありません．C関数のインポート機能とコールバック機能を組み合わせ，Pthreadライブラリをインポートすることで，SML#でネイティブなマルチスレッドプログラミングができます．SML#のコールバック機能は，C関数を呼び出したスレッドとは異なるスレッドからコールバックされたとしても，期待通りに動くように設計されています．そのため，SML#では，スレッドを生成する可能性のあるどのC関数も，ただそのままインポートし呼び出すだけで，スレッド生成機能を含めた全ての機能を，SML#から活用することができます．従って，Pthreadライブラリに限らず，例えば別スレッドで非同期的にコールバック関数を呼び出すサウンドプログラミングライブラリなども，SML#にインポートし，コールバックルーチンをSML#で書くことができます．もちろん，ユーザーが作成した独自のスレッドライブラリをSML#から使用することもできます．上述した`Pthread`ストラクチャも，Pthreadライブラリが定義する関数群をSML#にインポートして定義されています．

Pthread ライブラリをインポートして，SML#から使ってみましょう．まず，スレッドのハンドルの型 pthread_t に対応する SML#の型を決める必要があります．このマニュアルの執筆時点では，pthread_t は Linux では unsigned long int，macOS ではポインタと定義されています．どちらのプラットフォームでも，pthread_t はポインタと同じ大きさを持つ不透明な基本型と見なすことができますので，SML#では pthread_t を以下のように定義することにします．

```
type pthread_t = unit ptr
```

スレッドを生成する pthread_create 関数の型は，以下のようにインポートすることができます．

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

この関数を呼び出しスレッドを生成する関数 spawn を書きましょう．第 2 引数はスレッドの属性，第 4 引数はコールバック関数に渡す引数ですが，ここでは使用しないので，これらには NULL を渡すことにします．NULL ポインタは Pointer.NULL () で得られます．従って，spawn 関数は以下のように定義できます (エラー処理は省略します)．

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

同様に，スレッドの終了を待つ関数 pthread_join も，簡単にインポートできます．

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

このようにして定義した spawn と join を使ってこの節冒頭の例と同じプログラムを対話モードで書くと以下のようになります．

```
# 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
```

なお，

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

の行を

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

と書くことは危険です．なぜなら，ガベージコレクタが関数式の生成するクロージャを別スレッドで関数が呼び出される前に回収してしまう可能性があるからです．ガベージコレクションの影響については 29.2 節を参照してください．

11.2 MassiveThreads を用いた細粒度スレッドプログラミング

SML#は，MassiveThreads ベースの細粒度スレッドを提供します．MassiveThreads は，東京大学情報理工学系研究科で開発されている C 言語向けの軽量細粒度スレッドライブラリです．シームレスな C 言語との連携機構とスレッドを止めない並行 GC により，SML#は MassiveThreads を直接サポートします．MassiveThreads を利用することで，SML#プログラムから大量の（例えば 100 万個以上の）ユーザースレッドをマルチコア CPU 上で走らせることが可能です．

デフォルトでは，シングルスレッドプログラムの実行効率の調整のため，SML#からはただ 1 つのコアのみを使うように設定されています．マルチコア上での MassiveThreads を有効にするためには，`MYTH_`から始まる MassiveThreads 関連の環境変数を少なくとも 1 つ設定してください．例えば，対話モードの起動時に，

```
$ MYTH_NUM_WORKERS=0 smlsharp
```

などとして，環境変数 `MYTH_NUM_WORKERS` を定義してください．`MYTH_NUM_WORKERS` は，ユーザースレッドをスケジュールするワーカースレッドの数，すなわち使用する CPU コアの数を表します．`MYTH_NUM_WORKERS` が 0 のとき，Linux 環境ならば，全ての CPU コアを使用することを表します．

MassiveThreads ライブラリをほぼそのままバインドした `Myth` ストラクチャが標準で提供されています．`Myth.Thread` ストラクチャには，スレッドの生成と結合のための基本的な関数が含まれています．代表的な関数は以下の通りです．

- ユーザースレッドの起動．

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

  `create` f は f () を評価する新しいユーザースレッドを起動します．ユーザースレッドは MassiveThreads によって適切な CPU コアにスケジュールされます．スケジューリングポリシーはノンプリエンプティブです．つまり，一度あるスレッドがある CPU コア上で走り始めると，終了するか，スレッドの実行を制御する MassiveThreads ライブラリ関数（`Myth.Thread.yield`）を呼ばない限り，そのスレッドはその CPU コア上で走り続けます．

- ユーザースレッドの結合．

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

  `join` t はスレッド t の完了を待ち，t の評価結果を返します．`create` 関数で生成したユーザースレッドは，いつか必ず `join` されなければなりません．このストラクチャは MassiveThreads ライブラリを直接バインドしたものですので，多くの C ライブラリと同様に，生成したスレッドは明示的に解放されなければなりません．

- スレッドスケジューリング．

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

`yield()` は他のスレッドに CPU コアの制御を譲ります．

MassiveThreads の使い方を学ぶために，簡単なタスク並列プログラムを書いてみましょう．タスク並列プログラムは，おおよそ以下の手順で書くことができます．

1. 分割統治を行う再帰関数を書きます．
2. 再帰呼び出しのたびに新しいユーザースレッドが作られるように，再帰呼び出しを `create` と `join` で囲みます．
3. ただし，スレッドの計算コストがスレッドの生成コストを下回らないように，ある閾値（カットオフ）を下回った場合はスレッド生成をせず，同じスレッドで逐次的に再帰呼び出しをするようにします．逐次計算に切り替える閾値は，スレッド生成のオーバーヘッドよりも逐次処理時間が十分長くなるように決めます．経験的には，1 スレッドあたりおおよそ 3〜4 マイクロ秒程度の評価時間になるように設定するのが良いようです．

例えば，`fib 40` を再帰的に計算するプログラムは逐次的には以下のように書けます．

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

`fib (n - 1)` と `fib (n - 2)` が並列に計算されるように，その片方を `create` と `join` で囲むと，タスク並列のプログラムになります．

```
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
```

ただし，引数の `n` が十分に小さくなると，`fib n` の計算コストがスレッドの生成コストを下回ります．そこで，`n` が 10 を下回った場合は逐次で計算することにします．

```
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
```

これで並列 `fib` は完成です．このプログラムを実行すると，3,524,577 個のスレッドが生成されます．

第12章　SML#の拡張機能：SQLの統合

データを扱う実用的なプログラムを書くためには，データベースシステムとの連携が必要です．現在もっとも普及しているデータベースシステムは，問い合わせ言語 SQL を用いて操作します．種々のプログラミング言語で，データベース操作のためのマクロや関数が提供されていますが，データベースを使いこなす本格的なプログラムを書くためには，SQL 言語そのものを使いデータベースシステムを呼び出すコードを書く必要があります．これまでの方法は，SQL 文字列を生成するコードを書き，サーバに送ることでしたが，SML#では，SQL そのものを型を持つ（したがって第一級の）式として書くことができます．本節では，その利用方法を学びます．

12.1　関係データベースとSQL

現在の本格的なデータベースは，関係データモデルを基礎として実装された関係データベースです．SML#のデータベース連携を理解し使いこなすための準備として，まず本節で，関係データベースと SQL の基本を復習しましょう．

関係データモデルでは，データを関係の集まりとして表します．関係は，人の名前，年齢，給与などの複数の属性の関連を表し，通常以下のような属性名をラベルとして持つテーブルで表現します．

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

関係データベースは，これら関係の集まりを操作するシステムです．関係 R は，数学的には与えられた領域 $A_1, A_2, \cdots, A_n$ の積集合 $A_1 \times A_2 \cdots \times A_n$ の部分集合です．関係 R の要素 t は，n 個のタプル $(a_1, \ldots, a_n)$ です．実際のデータベースでは，タプルのそれぞれの要素にラベルを付け，レコードとして表現します．例えば上のテーブルの 1 行目の要素は，レコード {`name="Joe", age=21, salary=1000`} に対応します．これら関係に対して，和集合演算，射影（n 項関係を $m(\leq n)$ 項関係に射影），選択（集合の特定の要素の選び出し），デカルト積 $R \times S$ などの演算が定義されています．これら演算系を関係代数と呼びます．ここで留意すべき点は，関係モデルはレコードの集合に対する代数的な言語である点です．代数的な言語は，関数定義を含まない関数型言語です．

関係データベースでは，この関係代数を SQL と呼ばれる集合操作言語で表現します．SQL の中心は，以下の SELECT 式です．

`SELECT` $t^1.l_1$ `as` l'_1`,...,` $t^m.l_m$ `as` l'_m
`FROM` R_1 `as` t_1`, ...,` R_n `as` t_n
`WHERE` $P(t_1, \ldots, t_n)$

ここでは，以下の約束に従ってメタ変数を使用しています．

- R：関係を表す式．
- t：関係の中の一つのレコード要素を代表する変数
- l：属性名

- $t.l$：t の中の属性 l の値を表す式.

SELECT 式の意味は以下の操作と理解できます.

1. FROM 節で列挙された関係式 R_i を評価し，デカルト積 $R_1 \times \cdots \times R_n$ を作る.
2. デカルト積の任意の要素のタプルを $(t_1, \ldots, t_n)$ とする.
3. デカルト積から，WHERE 節で指定された述語 $P(t_1, \ldots, t_n)$ を満たす要素のみを選び出す.
4. 上記の結果得られた集合の各要素 $(t_1, \ldots, t_n)$ に対して，レコード $\{l'_1{=}t^1.l_1, \ldots, l'_m{=}t^m.l_m\}$ を構築する.
5. それらレコードをすべて集めた集合を，この式全体の結果とする.

例えば，上の例のテーブルを Persons とし，以下の SQL 式を考えてみましょう.

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

この問い合わせ式は以下のように評価されます.

- 関係 Persons のみのデカルト積は関係 Persons それ自身である.
- 関係 Persons の任意のタプルを P とする.
- P.Salary > 10000 の条件を満たすタプルを選び出し，以下の集合を得る.

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

- この集合の各要素 P に対して {name=P.name, age=P.age} を計算し，以下の集合を得る.

name	age
”Sue”	31
”Bob”	41

  このテーブルは，{{name="Sue", age=31}, {name="Bob", age=31}} のようなレコードのリストの表現と理解できる.

12.2　SML#への SQL 式の導入

SQL の SELECT 文は FROM 節に記述されたテーブルの集合から一つのテーブルを作成する式です．SQL 言語は，ある特定のデータベースへの接続の下で SQL 式を評価する機能を提供しますが，これをデータベース接続を受け取りテーブルを返す関数式に一般化すれば，関数型言語の型システムに統合することができます．テーブルはレコードの構造をしているため，多相レコードの型付けがほぼそのまま使用できます．しかし我々SML#チームの研究によって，このデータベースを受け取る関数の型付けには，多相レコード以外にさらに型理論的な機構が必要であることが示されています [14]．そこで，データベース接続を受け取る関数には特別の文法を用意します．SML#では，前節の SQL 式の例は，以下のような式で表現されます.

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

_sql db => ... は，データベース接続を変数 db として受け取る問い合わせ関数であることを示しています．_sql 式の中の>はデータベース問い合わせを実現するライブラリモジュール SQL の中に定義された SQL の値のための大小比較プリミティブです．#P.Name はタプル P の Name フィールドの取り出し演算，#db.Persons はデータベース db からの Persons テーブル取り出し演算です．これらは，SML#式では #Name P#Persons db と書かれるレコードからのフィールド取り出し式に相当します．_sql x => $expr$ 式では，SQL の文法に類似の構文を採用しています．この式に対して以下の型が推論されます．

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

この型は，'a の構造を持つデータベース接続の型'a conn から {age: 'c, name: 'e} を列とする結果テーブルの型への関数型です．'a は，この問い合わせに必要なデータベース構造を表すレコード多相型です．このように SQL 式は多相型を持つため，ML プログラミングの原理「式は型が正しい限り自由に組み合わせることができる」に従って，第一級のデータとして SML#の他の機能とともに自由にプログラムすることができます．

12.3 問い合わせの実行

_sql x => exp 式で定義された問い合わせ関数にデータベース接続を適用すれば，データベースをアクセスできます．そのために，以下の構文と関数が用意されています．

- データベースサーバー式．

  _sqlserver $serverLocation$: τ

  この式は，データベースサーバーを指定します．$serverLocation$ はデータベースサーバーの場所と名前です．その具体的な内容は，使用するデータベースシステムのデータベース指定構文に従います．τ はデータベースの構造です．テーブルの名前とその構造をレコード型の文法で指定します．この式を評価すると，τ SQL.server 型をもつデータベースサーバー定義が得られます．詳細は 22.3 節をご覧ください．

- データベースへの接続プリミティブ関数．

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

  SML#コンパイラは，τ SQL.server 型のデータベース定義を受け取り，その中に指定されたデータベースに接続しその構造をチェックし τ と一致していることを確認した後，τ SQL.conn の型をもつ τ 型のデータベースへの接続を返します．

- 問い合わせの実行．_sql 式そのものが，データベースで問い合わせを実行する関数を表します．データベース問い合わせ関数にデータベース接続を適用すると，その問い合わせがその接続を通じてデータベース上で実行され，その結果にアクセスするための τ SQL.cursor 型のカーソルを返します．

- 問い合わせ結果の取得．以下の 2 つの手続き的な関数が提供されています．

```
SQL.fetchAll : ['a. 'a SQL.cursor -> 'a option]
SQL.fetch : ['a. 'a SQL.cursor -> 'a list]
```

SQL.fetch は先頭のレコードを返し，カーソルを次のレコードに進めます．SQL.fetchAll は，問い合わせによって得られた関係をすべて読み込み，リストに変換します．

- 問い合わせの後処理．

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

SQL.closeCursor は問い合わせ処理の終了を，SQL.closeConn はデータベース接続の終了をそれぞれデータベースサーバに通知します．

12.4 データベース問い合わせ実行例

以上の構文を使いデータベースの問い合わせを行なってみましょう．SML#の通常のインストールでは，データベースサーバは PostgreSQL と接続するように設定されています．データベースをアクセスするためには，PostgreSQL サーバをインストールし起動しておく必要があります．

まず以下の手順で，PostgreSQL サーバでデータベースを構築しましょう．以下に簡単な手順を示します．詳しくは PostgreSQL ドキュメントを参照してください．

1. PostgreSQL サーバを起動します．例えば pg_ctl start -D /usr/local/pgsql/data ようにすると起動できるはずです．
2. コマンドラインで createuser *myAccount* を実行し PostgreSQL のユーザのロールを作成する．*myAccount* は，使用するユーザの名前です．
3. コマンドラインで createdb mydb を実行しデータベースを作成する．
4. SQL 言語インタープリタを起動し，データベースの中にテーブルを作成する．例えば，第 12.1 節の例のデータベースは，以下のようにすれば作成できます．

```
$ 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);
```

自分のアカウント（ここでは *myAccount*）に戻り，データベースにアクセスできるか確認してみましょう．以下のような結果が得られれば，成功です．

```
$ psql mydb
mydb=# SELECT * FROM Persons;
mydb=# SELECT * FROM Persons;
 name | age | salary
------+-----+--------
```

```
 Joe  |  21 |  10000
 Sue  |  31 |  20000
 Bob  |  41 |  20000
(3 行)
```

ではいよいよ，このデータベースを SML#からアクセスしてみましょう．第 12.2 節で定義した問い合わせ関数を myQuery とします．対話型セッションで以下のような結果が得られるはずです．

```
$ 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 その他の SQL 文

SQL 言語は，データベースからデータを取り出す select 文以外に，様々な機能が提供されています．SML#の 3.7.1 版では，以下の SQL コマンドをサポートしています．

- 問い合わせ (SELECT)
- タプルの追加と削除 (INSERT, DELETE)
- テーブルの更新 (UPDATE)
- トランザクションの実行 (BEGIN, COMMIT, ROLLBACK)

SELECT クエリでは，基本的な加え，以下の機能が使えます．

- 自然結合 (NATURAL JOIN)
- 内部結合 (INNER JOIN)
- 行集約 (GROUP BY, HAVING)
- サブクエリ，相関サブクエリ，EXISTS サブクエリ
- ソート (ORDER BY)
- 行数制限 (LIMIT, OFFSET, FETCH)

SQL 構文の詳細は 22 章をご覧ください．SML#開発チームはより完全な SQL 言語の統合を目指して機能の追加を続けています．

第13章　SML#の拡張機能：動的型付け機構とJSONの型付き操作

ネットワーク通信やファイル入出力など，データをプログラムの外部とやりとりしたりするときには，データをある形式の文字列に変換します．例えば，JSON は，今日のインターネットで広く普及しているデータ形式のひとつです．文字列に変換されたデータは，ML のデータとは異なり，型の制約がありません．そのため，文字列に変換されたデータを扱おうとすると，データ構造に合った型を付けられない（例えば，レコード構造を持っているにもかかわらずレコード型を付けられない）ことがあります．SML#の動的型付け機構は，文字列化データなど動的に構築された値と，静的に型付けされた ML の値とを相互変換する機能を提供します．SML#はさらに，この機構を通じて，JSON が表現するデータ構造に ML のレコード型に近い型を与え，静的に型付けされた JSON 操作プリミティブを提供します．本章では，その利用方法を学びます．

13.1　動的型付け

動的型付き言語では，値の型を実行時に分析し，その型に応じた処理を行うことができます．動的型付き言語における値は，型を表す実行時表現とその型に応じた値とを持つ，一様な形式のデータとみなせます．SML#は，この型情報を伴う値の表現を，静的型付き言語である ML から取り扱う機構を提供します．

`Dynamic.void Dynamic.dyn` は，動的に型付けされた値の型です．型引数 `Dynamic.void` の意味は後述します．この型に対して，以下の基本的な演算が定義されています．

- 関数 `Dynamic.dynamic : ['a#reify. 'a -> Dynamic.void Dynamic.dyn]`．この関数は任意の ML の値から動的型付けされた値を作ります．`'a#reify` は，`'a` のインスタンスの実行時型情報が必要であることを表します．他のカインドと異なり，`reify` カインドは型変数が動く範囲を制限しません．
- 式 `_dynamic` exp `as` τ．この式は，動的型付けされた値 exp の型情報を実行時に検査し，その値を τ 型にキャストします．キャストに失敗した場合は実行時例外 `Dynamic.RuntimeTypeError` が発生します．

さらに以下の式が用意されています．

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

この式は，動的型付けされた値 exp に対して動的型検査を伴うパターンマッチを行います．パターン $dpat_i$ には，任意のパターンを書くことができます．ただし，変数パターンには必ず型注釈を書く必要があります．

以下の対話セッションは，ヘテロジニアスなリストとそれを処理する関数の例です．

```
# 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
```

レコードに限らず，データ型，関数，不透明な型など任意の型の値を動的型付けされた値にしたり，静的型付けされた値に戻したりすることができます．

13.2 項と型のリーフィケーション

静的型付き言語では一般に，型情報はコンパイラのみが管理するメタな情報であり，型情報はコンパイル後のコードにはデータ構造として含まれません．また，値のメモリ上のデータ構造も，型によって決められるメタな情報です．これらメタな情報をコードが扱えるオブジェクトとして取り出すことをリーフィケーションと言います．SML#の動的型付けは，このリーフィケーションを基盤として構築されています．ユーザーもこのリーフィケーション機構を利用して，型情報や値の内部構造を，ML の datatype として取り出すことができます．

SML#は以下の関数を提供します．

- `Dynamic.dynamicToTerm : Dynamic.void Dynamic.dyn -> Dynamic.term`．この関数は，動的に型付けされた値から値の構造を `Dynamic.term` 型の項表現として取り出します．`Dynamic.term` 型は一般的な ML の datatype であり，他の datatype と同様，ML の `case` 式などで分析することができます．

- `Dynamic.dynamicToTy : Dynamic.void dyn -> Dynamic.ty`．この関数は，動的に型付けされた値から型情報を `Dynamic.ty` 型の項表現として取り出します．`Dynamic.term` 型と同様に，`Dynamic.ty` 型もごく一般的な ML の datatype です．

- `Dynamic.termToDynamic : Dynamic.term -> Dynamic.void Dynamic.dyn`．この関数は項表現を動的に型付けされた値に変換します．`_dynamic` 構文と組み合わせることで，`Dynamic.term` 型の値として構築された構造を，ML の静的型付けされた値に変換することが可能です．

以下の対話セッションは，`Dynamic.term` 型を通じて ML のレコードから別の型のレコードを構築する例です．

```
# 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 プリティプリンタ

SML#の対話セッションで値を表示する機能も，リーフィケーション機構の上に実現されています．この対話セッションのプリンタを，任意の ML データをプリティプリントする関数としてユーザープログラムから呼び出すことができます．以下の関数が提供されています．

- `Dynamic.pp : ['a#reify. 'a -> unit]`．この関数は任意の ML データを対話セッションと同じ形式で標準出力に出力します．プリントデバッグなどに有用です．
- `Dynamic.format : ['a#reify. 'a -> string]`．標準出力に出力するかわりに文字列を返します．

これらの関数は多相関数内からも用いることができます．ただし，多相関数で用いた場合，その多相関数の型に `reify` カインドが付いてしまい，型が変わってしまうことがあることに注意が必要です．

13.4 JSON と部分動的レコード

JSON は，整数や文字列などの基本データ型，データをラベルに対応づけるオブジェクト型，および任意のデータを並べた配列型からなるデータ構造です．これらのデータ構造はそれぞれ，ML の基本型，レコード型，リスト型に対応します．

しかし，ML のそれらとは異なり，JSON には型の制約がありません．JSON オブジェクトでは，同じラベルに同じ型のデータが入っているとは限りませんし，そもそもそのラベルが常にあるとも限りません．また，JSON の配列は，全ての要素の構造が一致する必要はありません．実際に JSON が利用される局面においても，このようなヘテロジニアスなデータが頻繁に現れます．そのため，JSON に ML の型をそのまま付けることは難しく，また ML の型が付くものに限定することは JSON の実態に合っていません．

SML#では,「動的部分レコード」と呼ばれる考え方に基づき，SML#プログラム中で JSON データに型を与えます．動的部分レコードとは，その構造の一部のみが静的に分かっていて，それ以外の構造は実行時チェックをしなければ分からないようなレコード構造のことです．SML#では，実行時に得られた JSON データを動的部分レコードとして取り扱います．

SML#での型付き JSON 操作を，例を通じて見てみましょう．SML#では，初めて読み込まれた JSON データは全て `Dynamic.void Dynamic.dyn` 型を持ちます．この `Dynamic.void` は，データ構造が静的には一切分かっていないことを表します．例えば，

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

という JSON は，まず Dynamic.void Dynamic.dyn 型の値として読み込まれます．

この JSON を受け取るプログラムは，受け取った JSON は少なくとも文字列型の name フィールドを持つことを期待していて（持っていなければ実行時例外とする），そのフィールドの値を取り出したいとします．そのために，プログラムは受け取った JSON データに対して動的型検査を行い，文字列を持つ name フィールドが存在することを確認します．この検査が成功すれば，その JSON は少なくとも文字列型の name フィールドがあることが確認されているはずです．この検査の結果に対し，SML#は，このことが確認されたことを表す型 {name : string} Dynamic.dyn を静的に与えます．

部分的に型が判明している JSON から値を取り出すには，以下の関数を用いて，その静的なビューを取り出します．

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

この関数は，動的部分レコードである JSON データのうち，静的に判明している構造のみを，ML のデータ構造に変換する関数です．この関数を用いることで，{name : string} Dynamic.dyn 型の JSON データから，{name : string} 型のレコードを取り出すことができます．

JSON として受け取ったデータは，もはや ML のデータに変換されました．これ以降の操作は，ML の言語機能を自由に組み合わせて型安全に書くことができます．

13.5 JSON の操作

SML#では以下の構文および関数を提供します．

- JSON の読み込み．

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

  この関数は，JSON 文字列を構文解析し，動的部分レコードとして読み込みます．構文解析に失敗した場合は Dynamic.RuntimeTypeError 例外を発生させます．Dynamic.void は，値の構造が一切判明していないことを表します．

- JSON の動的型検査_dynamic *exp* as τ および動的型検査を伴うパターンマッチ_dynamiccase 式．13.1 節で述べたこれらの式は JSON の検査にもそのまま使えます．τ には以下の型を書くことができます．

  - int，bool，string などの基本型，レコード型，リスト型，およびこれらの組み合わせ．JSON データが τ が表す構造に一致するときのみ，キャストは成功します．
  - 部分動的レコード型 $\{l_1{:}\tau_1, \cdots, l_n{:}\tau_n\}$ Dynamic.dyn．JSON データが少なくとも $l_1, \ldots, l_n$ の n 個のラベルを持ち，各ラベルの値が $\tau_1, \ldots, \tau_n$ にキャストできるとき，キャストは成功します．
  - 構造が判明していない型 Dynamic.void Dynamic.dyn．この型へのキャストは常に成功します．
  - 以上の型を自由にネストさせた組み合わせ．

- 静的なビューの取得．

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

  この関数は，与えられた JSON データのうち，静的に構造が判明している部分のみを ML のデータに変換します．引数が Dynamic.void Dynamic.dyn 型のときは Dynamic.RuntimeTypeError 例外が発生します．

- JSON プリンタ.

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

  この関数は，与えられた JSON データの文字列表現を返します．文字列表現には，'a のインスタンスによらず，JSON データに含まれる全体が含まれます．

その他，動的型付けやリーフィケーションの諸機能も，型が示すように，そのまま JSON に用いることができます．逆に，動的型付けされた値を部分動的レコードにすることも可能です．

13.6 JSON プログラミング例

以下の対話セッションは，ML のレコードのリストに対応する構造を持つ JSON を読み込む例です．

```
# val J = "[{n̈amë:J̈oe̊, ägë:21, g̈radë:1.1},"
            ^ "{n̈amë:S̈ue̊, ägë:31, g̈radë:2.0},"
            ^ "{n̈amë:B̈ob̊, ägë:41, g̈radë: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
```

より複雑な例を見てみましょう．以下のような JSON を考えます．

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

この JSON 文字列が変数 J に束縛されているとします．これはヘテロジニアスなリストです．このようなリストは，直接 ML のリストにキャストすることができません．一方，このリストを，各要素の共通部分に注目し,「少なくとも name と age フィールドを持つレコードのリスト」と見ることはできます．この見方に基づき，以下のようにして J を読み込みます．

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

vl の型は {name:string, age:int} Dynamic.dyn list です．このリストの静的なビューは，

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

として取り出すことができます．さらに，各要素から name フィールドの値を取り出したければ，

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

とします．

リストの各要素に対して，あるフィールドの有無をチェックしたい場合は，個々の要素に対して動的検査を行います．例えば，nickname がある場合はそれを，そうでなければ name を取り出したい場合は，以下のように書きます．

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

この getFriendlyName 関数の型は，['a#reify#{name : string}. 'a Dynamic.dyn -> string] です．多相レコード型と動的レコード型が組み合わさっていることに注意してください．これは，JSON を処理する観点からは，直感的には {name : string} Dynamic.dyn -> string と同様の意味を持つように見えますが，これらの厳密な意味は異なります．後者は，name 以外のフィールドの存在は動的型検査しなければ分からないものだけしか引数に取れないのに対し，前者は name 以外のフィールドの存在が確認されているものも引数に取ることができます．

第14章　SML#の拡張機能：SML#分割コンパイルシステム

実用言語としてのSML#の大きな特徴は，完全な分割コンパイルのサポートです．以下の手順で大規模なプログラムを開発していくことができます．

1. 個々のプログラムモジュールのオブジェクトファイルへのコンパイル
2. オブジェクトファイルを，C言語のオブジェクトファイルやシステムライブラリとともにリンクし実行形式ファイルを作成する．

さらにSML#コンパイラは，各オブジェクトの依存関係をシステムの `make` コマンドが解釈できる形式で出力することができます．この機能を使用すれば，C言語などとともに大規模システムを効率よく開発することが可能です．本章では，この分割コンパイルシステムの概要を説明します．

14.1　分割コンパイルの概要

分割コンパイルシステムは以下の手順で使用します．

1. システムを構成するソースをコンパイル単位に分割する．この分割は，どのような大きさでもよく，分割された各々の機能は，SML#で記述可能な宣言で実現できるものであれば，特に制約はありません．ここでは，システムを `part1` と `part2` に分割したとします．
2. 分割した各要素のインターフェイスを定義する．このインターフェイスは，第14.3節で説明するインターフェイス言語で記述します．システムを `part1` と `part2` に分割するシナリオでは，まず，インターフェイスファイル `part1.smi` と `part2.smi` を用意します．
3. 各インターフェイスファイルに対して，それを実現するソースファイルを開発し，それぞれのソースファイルをコンパイルしオブジェクトファイルを作成する．

   上記シナリオの場合，`part1.smi` と `part2.smi` を実現するソースファイル `part1.sml` と `part2.sml` を開発し，コンパイルしオブジェクトファイル `part1.o` と `part2.o` を作成します．

   この開発は，それぞれ完全に独立に行うことができます．ML系言語の場合，ソースファイルの開発では，コンパイラによる型推論が威力を発揮します．分割コンパイルシステムは，その名のとおりそれぞれ独立にコンパイルできます．例えば，`part2` が `part1` の関数などを使用する場合でも，`part1.sml` を開発する以前に，`part2.sml` をコンパイルできます．この機能により，システムのどの部分でも，MLの型推論の機能をフルに活用し，開発を行うことができます．
4. オブジェクトファイル集合を必要なライブラリと共にリンクし，実行形式ファイルを作成する．

   例えば，`part1` と `part2` 以外に，それらから参照されるC関数のファイルやC言語で書いた独自のライブラリを使用する場合，それらの場所をコンパイラに指示することによって，それらとともにをリンクしOS標準の実行形式ファイルを生成できます．なお，SML#コンパイラは，SML#ランタイムの実行に必要なC言語の標準ライブラリを常にリンクするため，標準ライブラリは指定なしに使用することができます．

より本格的かつ高度なシナリオは以下のようなものが考えられます.

1. システムを C 言語で書く部分と SML#言語で書く部分に分割する.
2. C 言語の部分は，ヘッダファイル（.h ファイル）とソースファイル（.c ファイル）を開発する.
3. SML#言語の部分は，インターフェイスファイル（.smi ファイル）とソースファイル（.sml ファイル）を開発する.
4. SML#コンパイラの依存性出力機能を利用し，Makefile を作成する.
5. システムを `make` コマンドを用いて，コンパイルとリンクを行う.

SML#コンパイラ自身，C 言語と SML#で書かれていて，この手順で種々のツールを含むシステムがコンパイルリンクされ，SML#コンパイラが作成されます.

14.2　分割コンパイル例

前節のシナリオに従い，乱数を使うおみくじプログラムを例に，分割コンパイルをして実行形式ファイルを作ってみましょう．システムを

- `random`：乱数発生器
- `main`：メインパート

に分割することにします．まず，このインターフェイスファイルを以下のように設計します.

- `random.smi` の定義.

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

  このインターフェイスファイルは，このインターフェイスファイルを実装するソースファイルが，他のファイルは参照せずに `Random` structure を提供することを表しています.

- `main.smi` の定義.

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

  このインターフェイスファイルは，SML#の基本ライブラリ`"basis.smi"`（The Standard ML Basis Library）とこのディレクトリにある `random.smi` を利用し，外部には何も提供しないことを意味しています.

このインターフェイスを使えば，`main.sml` と `random.sml` を独立に開発できます．`main.sml` は図 14.1 のように定義できます．このファイルは，`random.smi` を実装するソースファイルが存在しなくても，コンパイルしエラーがないか確認することができます.

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

-c は SML#コンパイラにコンパイルしオブジェクトファイルを生成を指示します．対話型での使用と同様に型チェックをした後，エラーがなければ，オブジェクトファイル main.o を作ります．インターフェイスファイルは，ファイル名の .sml を .smi に変えたものが自動的に使用されます．ソースコード冒頭に _interface *filePath* 宣言を書くことによりインターフェースファイルを明示的に指定することもできます．

次に，random.sml を開発します．高品質の乱数発生関数の開発は，数学的な知識と注意深いコーディングが要求されます．ここでは，これはスクラッチから開発するのではなく，既存の C での実装を使うことにします．種々ある乱数発生アルゴリズムの中で，その品質と速度の両方の点から Mersenne Twister が信頼できます．このアルゴリズムは C のソースファイル mt19937ar.c として提供されています．

そこでこのファイルをダウンロードしましょう．インターネットで Mersenne Twister あるいは mt19937ar.c をサーチすれば簡単に見つけることができます．このファイルには以下の関数が定義されています．

```
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);
```

この中の main は，このアルゴリズムをテストするためのメイン関数です．我々は新たな実行形式プログラムを作成するので，main 関数は，我々のトップレベルファイル main.sml をコンパイルしたオブジェクトファイルに含まれているはずです．そこで，mt19937ar.c ファイルの関数 int main(void) の定義をコメントアウトする必要があります．その他の関数は，SML#から利用できるライブラリ関数です．ここでは以下の２つを使うことにします．

- void init_genrand(unsigned long s)．シーズの長さを受け取りアルゴリズムを初期化する関数です．シーズ s は非負な整数ならなんでも構いません．
- long genrand_int31(void)．初期化された後は，呼ばれる毎に 31 ビットの符号なし整数（32 ビット非負整数）のランダムな列を返します．

そこで，これを利用して，random.sml を以下のように定義します．

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

このソースファイルも，以下のコマンドにより，このファイルだけで単独にコンパイルできます．

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

このソースファイルの定義と並行して，(main 関数をコメントアウトした）Mersenne Twister をコンパイルし，オブジェクトファイルを生成しておきます．

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

以上ですべてのソースファイルがそれぞれコンパイルされ，オブジェクトファイルが作られたはずです．それらオブジェクトファイルは，トップレベルのインターフェイスファイルと必要なオブジェクトファイルを指定することによって，実行形式ファイルが作成されます．

```
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 "好きな数を入力してください（0 で終了です）. ";
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();
```

図 14.1: main.sml の例

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

SML#コンパイラは，smi ファイルを解析し，このファイルから参照されている smi ファイルを再帰的にたどり，対応するオブジェクトファイルのリストを作り，コマンドラインに指定された C 言語のオブジェクトファイルと共に，システムのリンカーを起動し，実行形式ファイルを作成します．

14.3 インターフェイスファイルの構造

インターフェイスファイルは，分割コンパイルするコンパイル単位のインターフェイスを記述したファイルです．インターフェイスファイルの内容は，Require 宣言と Provide 宣言からなります．Require 宣言は，コンパイル単位が使用する他のコンパイル単位を以下の形の宣言として列挙します．

```
_require smiFilePath
```

smiFilePath は，他のコンパイル単位のインターフェイスファイル（.smi ファイル）です．よく使用されるインターフェイスの集合には，それらをまとめた名前が付けられ，システムに登録されています．代表的なものは，The Definition of Standard ML Basis Library で実装が義務付けられている基本ライブラリのインターフェイスファイルをすべて含む basis.smi です．プログラムのインターフェイスファイルの冒頭に

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

queue.smi ファイル：

```
_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 ファイル:

```
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
```

図 14.2: インターフェイスファイルの例

と書いておくと，基本ライブラリが使用可能となります．

Provide 宣言は，コンパイル単位が他のコンパイル単位に提供する SML#の資源を記述します．記述する資源は，おおよそ，Standard ML のソースファイルの宣言定義できるものすべてと考えるとよいでしょう．具体的には，以下のものが含まれます．

- `datatype` 定義．
- `type` 定義．
- `exception` 定義．
- `infix` 宣言．
- 変数とその型の定義．
- モジュールの定義
- `functor` の定義

キューのインターフェイスファイル `queue.smi` とそのインターフェイスを実装する `queue.sml` の例を図 14.2 に示します．

`queue.smi` ファイルの冒頭に書かれている`_require "basis.smi"`は，

- この queue.smi ファイルが基本ライブラリを使うこと，
- この queue.smi ファイルの中で定義されずに使われている'a list などの型は，基本ライブラリに定義されていること

を表しています．それ以降の定義が Provide 宣言部分です．この例では，このインターフェイスを実装するモジュールが Queue という structure を提供することを示しています．この例から理解されるとおり，インターフェイスファイルの Provide 宣言は Standard ML のシグネチャと似た構文で記述します．しかしシグネチャと考え方が大きく違うのは，datatype や exception などは仕様ではなく，実体に対応するということです．queue.smi の中で定義される datatype 'a queue と exception Dequeue は，queue.sml で定義される型および例外そのものとして扱われます．従って，この queue.smi を複数のモジュールで_require 宣言を通じて利用しても，queue.smi で定義された同一のものとして扱われます．

14.4 型の隠蔽

前節の例から理解されるとおり，インターフェイスファイルの基本的考え方は，以下のとおりです．

1. datatype や exception などのコンパイル時に定義される資源（静的資源）は，将来そのインターフェイスファイルで定義される実体そのものを記述する．
2. 関数や変数などの実行時の値を表すものは，その型のみを宣言する．

これら情報が，コンパイラがこのインターフェイスファイルを使う別のソースコードをコンパイルする上で必要かつ十分な情報です．しかし，この原則だけでは，Standard ML のモジュールシステムが提供する型情報の隠蔽の機能を使うことができません．例えば，前節のインターフェイスファイル queue.smi では，'a queue の実装が，Q of 'a list * 'a list と定義され公開されていますが，この実装の詳細は隠蔽したい場合が多いと思われます．

この問題の解決のために，インタフェイスファイルに

```
type tyvars tyid (= typeRep)     (* 括弧はそのまま記述する *)
eqtype tyvars tyid (= typeRep)     (* 括弧はそのまま記述する *)
```

の形の隠蔽された型の宣言を許しています．この宣言は，型 *tyid* が定義されその実装の表現は *typeRep* であるが，その内容はこのインターフェイスの利用者からは隠されることを表しています．シグネチャ同様，type 宣言は，同一性判定を許さない型，eqtype は同一性判定が可能な型を表します．*typeRep* には，*tyid* を実装する型の型コンストラクタを指定します．例えば，

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

という実装に対して，インターフェースファイルは

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

と書きます．実装型がレコード型，組型，関数型ならば，*typeRep* にはそれぞれ {}，*，->を指定します．実装型が datatype で定義された型ならば，コンストラクタの定義に応じて以下のいずれかを指定します．

- unit．引数を持たないコンストラクタだけからなり，かつコンストラクタが 1 つしか存在しない．
- contag．引数を持たないコンストラクタだけからなり，かつコンストラクタが 2 つ以上存在する．

- boxed．上記のいずれにも当てはまらない．

例えば，queue.smi の datatype 'a queue 宣言のかわりに

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

と書くことができます．

さらに，シグネチャの場合同様，インターフェイスファイルは，そのインターフェイスを実装するソースが定義する変数をすべて網羅している必要はありません．インターフェイスに宣言されたもののみが，そのインターフェイスを_require 宣言を通じて利用するユーザに見えるようになります．

14.5 シグネチャの扱い

Standard ML 言語では，第 14.3 節で説明した資源以外に，シグネチャも名前の付いた資源です．例えば，Queue ストラクチャに対しては，QUEUE シグネチャが定義されていると便利です．SML#のインタフェイスの Require 宣言には，以下の構文により，シグネチャファイルの使用宣言も書くことができます．

```
_require sigFilePath
```

sigFilePath はシグネチャファイルのパス名です．この機構を理解するために，Standard ML のシグネチャの以下の性質を理解する必要があります．

- シグネチャは，他の structure で定義された型（の実体）の参照を含むことができる．
- シグネチャ自身は，型（の実体）を生成しない．

SML#コンパイラは，_require *sigFilePath* 宣言を以下のように取り扱います．

- シグネチャ以外のすべての Require 宣言の下で，*sigFilePath* ファイルで定義されたシグネチャを評価．
- この宣言を含むインターフェイスファイルを_require 宣言を通じて利用するソースコードの冒頭で，評価済みの *sigFilePath* ファイルが展開されているとみなす．

図 14.3 に，シグネチャを含み型が隠蔽された待ち行列のインタフェイスと実装の例を示します．この例では，実装ファイルに型を隠蔽するシグネチャ制約を付けても，利用者にとっての効果は変わりません．

14.6 ファンクタのサポート

SML#の分割コンパイルシステムは，ファンクタもオブジェクトファイルに分割コンパイルし，他のコンパイル単位から_require 宣言を通じて利用することができます．ファンクタのインターフェイスファイルは，その Provide 宣言に以下のように記述します．

```
functor id(signature) =
struct
  (* この部分は structure の Provide と同一 *)
end
```

ここで，*signature* は Standard ML 構文規則に従うシグネチャ宣言です．インターフェイスと似ていますが，インターフェイスではなく，通常のシグネチャが書けます．以下に二分探索木を実現するファンクタのインターフェイスファイルの例をしめします．

queue-sig.sml ファイル：

```
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 ファイル：

```
_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 ファイル:

```
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
```

図 14.3: インターフェイスファイルの例

```
_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
```

ただし，この機構を利用するプログラマは，以下の点に留意する必要があります．

- functor はモジュール分割のための道具ではない．分割コンパイルができない ML 系言語処理系では，モジュールの間の直接の依存関係を断ち切る手段としてのファンクタの使用が推奨されることがありました．例えば，

  A.sml ファイル：

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

  B.sml ファイル：

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

  と書くと `B.sml` ファイルが別な実装ファイル `A.sml` ファイルに直接依存してしまいます．この `B.sml` をファンクタを使い以下のように書きなおせば依存性は解消されます．

  B.sml ファイル：

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

  この機能は，分割コンパイルの機能そのものです．分割コンパイルとリンクの機能を完全にサポートしている SML# では，この目的のためにファンクタを使う必要はなく，このような使用は避けるべきです．

- ファンクタの利用にはコストがかかる．ファンクタは，関数などの値以外に型もパラメタとして受け取る能力があります．これが，ファンクタなしでは達成できないファンクタ本来の機能です．しかし，同時に型をパラメタとして受け取り，その型に応じた処理を行うため，ファンクタ本体のコンパイルには，型が決まっているストラクチャに比べてコンパイルにもコンパイルされたオブジェクトコードにもオーバヘッドが生まれます．これは，高度な機能を使用する上で避けられないことです．ファンクタは，このオーバヘッドを意識して，コードすべき高度な機能です．

現在の SML#のファンクタの実装には以下の制限があります．

- ファンクタの引数に型引数を持つ抽象型コンストラクタが含まれている場合，その型コンストラクタにはヒープにアロケートされる実装表現（array, boxed, {}, ->, *など）を持つ型のみを適用することができます．例えば，SML#では以下の例はコンパイルエラーになります．

```
# 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　レプリケーション宣言

インタフェイスファイルは，資源の実体の表現です．従って，例えば

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

のような構造のモジュールに対して，シグネチャと違い，

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

のような書き方はできません．これまでのメカニズムのみでは，Bar の内容をくり返し書く必要があります．この冗長さを抑止するために，インタフェイスファイルは，以下の資源のレプリケーション（複製）宣言を許しています．

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

これによって，すでに定義済みの資源に対しては，インタフェイスファイルであっても，通常のソースファイルと同様にその複製の宣言を書くことができます．例えば，Bar ストラクチャを提供するインタフェイスファイルが bar.smi であるとすると，以下のように書くことができます．

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

14.8 トップレベルの実行

SML#プログラムの実行は，.sml ファイルのトップレベルに書かれた val 宣言を上から順に評価することで行われます．ひとつのプログラムが複数の.sml ファイルからなるとき，どのファイルのトップレベルがどの順番で実行されるかは，自明ではありません．SML#では，各.sml ファイルのトップレベルの実行順序を，以下の規則に従って決定します．

1. リンク時に smlsharp コマンドに指定した.smi ファイルに対応する.sml ファイルのトップレベルは必ず最後に実行されます．この条件は以下のどの条件よりも優先されます．以下，この条件に合致する.sml ファイルのトップレベルをリンクトップレベルと呼びます．

2. A.sml が Provide している変数を B.sml が使用しているとき，A.sml のトップレベルは B.sml のトップレベルより先に実行されます．

3. 定義-使用関係に無い 2 つの.sml ファイルのトップレベルの実行順序は不定です．

4. A.sml が何も Provide していないか，あるいはリンクトップレベルから変数の定義-使用関係を辿って A.sml に到達できない場合，A.sml のトップレベルは実行されません（SML#コンパイラは A.o をリンクしません）．

トップレベルの実行順序およびリンクされるオブジェクトファイルの集合は，_require によって決まるのではなく，ファイル間での変数の定義-使用関係によって決まることに注意してください．たとえ A.smi が他の.smi ファイルから _require されていたとしても，A.smi が Provide する変数をどの.sml ファイルも使用していなければ，A.sml のオブジェクトファイル A.o はリンクされません．従って，例えば print しかしない.sml ファイルなど，副作用のあるトップレベルコードだけからなり何も Provide しない.sml ファイルは，リンクも実行もされません．

例外は 2 つあります．ひとつはリンクトップレベルです．リンクトップレベルは必ず実行されます．もうひとつは，Require 宣言を

_require *smiFilePath* init

と書くことです．A.smi に _require "B.smi" init と書かれている場合，変数の定義-参照関係にかかわらず，B.smi のトップレベルが A.smi のトップレベルより先に実行されます．ただし，この書き方は特殊な用途に使用すべきものであり，常用は推奨されません．従って，プログラムの初期化時に必ず実行されなければならない print や配列の破壊的更新は，リンクトップレベルで行うか，リンクトップレベルから呼び出される関数で行うようにしてください．

第III部

参照マニュアル

第15章　序論

第III部では，言語，ライブラリ，`smlsharp`コマンド，システム及び外部インターフェイスを含むSML#のシステム仕様を定義する．第I部で述べた通り，SML#はThe Definition of Standard ML[5]と後方互換性のある言語である．The Definition of Standard MLには語彙と文法の定義に加え，静的意味（型チェック仕様）および動的意味（評価の仕方）が意味関係を導く導出システムとして定義されている．しかし，SML#は，それ自身で完結した謂わば閉じた系であるStandard MLと異なりCとの直接連携，SQLのシームレスな統合等を含むオープンな系であり，これらの機能を含む形式的かつユーザの理解に貢献するような簡潔な定義手法は未だ確立されていない．そこで，本参照マニュアルでは，言語の型の型の性質と意味を，必要に応じて，日本語による注釈として与えることにする．

15.1　使用する表記法

構文構造の定義あたって，以下の表記を用いる．

- 終端記号は，タイプライタフォントを用いて`"SML#"`のように記述する．
- 非終端記号や構文上の名前は$\langle exp \rangle$のように鍵カッコ書こんで記述する．
- 省略可能な要素は(opt)?と書く．
- 各構文のクラス$\langle x \rangle$に対して，$\langle xList \rangle$は，空白で区切られた$\langle x \rangle$の1個以上の列を表し，$\langle xSeq \rangle$は以下のような構造の何れかを表す．

$\langle xSeq \rangle$	::=	$\langle x \rangle$	要素1個
	|		空列
	|	$(\langle x \rangle_1, \cdots, \langle x \rangle_n)$	n個の組

第16章　SML#の構造

本章では，SML#プログラムの構造，型の概念，実行のモデル等を含む言語構造を解説する．

16.1　対話型モードのプログラム

対話型モードのSML#プログラムは，セミコロン (;) で終了する宣言の列である．以下はインタラクティブセッションの例である．

```
$ smlsharp
SML# 3.7.1 ...
# fun fact 0 = 1
>   | fact n = n * fact (n - 1);
> val x = fact 10;
val fact = fn : int -> int
val x = 3628800 : int
```

ここで，#と>は，対話型SML#コンパイラが印字する先頭行及び継続業のプロンプト文字である．この例のように，対話型コンパイラは，入力された宣言の評価結果を表示する．

宣言には，関数やレコードなどの値を生成するプログラムである核言語の宣言 ⟨*decl*⟩ と，それら宣言をひとまとまりにして名前をつけるモジュール言語の宣言 ⟨*topDecl*⟩ に大別される．

$$
\begin{array}{lll}
\langle interactiveProgram \rangle & ::= & ; \\
& | & \langle decl \rangle \ \langle interactiveProgram \rangle \\
& | & \langle topdecl \rangle \ \langle interactiveProgram \rangle
\end{array}
$$

- 核言語の宣言

  以下に核宣言 (⟨*decl*⟩) の構造と簡単な例を示す．

⟨*decl*⟩	::=	⟨*infixDecl*⟩	infix 宣言
	|	⟨*valDecl*⟩	val 宣言
	|	⟨*valRecDecl*⟩	val rec 宣言
	|	⟨*funDecl*⟩	関数宣言
	|	⟨*datatypeDecl*⟩	データ型宣言
	|	⟨*typeDecl*⟩	型の別名宣言
	|	⟨*exceptionDecl*⟩	例外宣言
	|	⟨*localDecl*⟩	局所宣言

宣言の種類	簡単な例
infix 宣言	`infix 4 =`
val 宣言	`val x = 1`
val rec 宣言	`val rec f = fn x => if x = 0 then 1 else x * f (x - 1)`
関数宣言	`fun f x = if x = 0 then 1 else x * f (x - 1)`
データ型宣言	`datatype foo = A | B`
型宣言	`type person = {name:string, age:int}`
例外宣言	`exception Fail of string`
局所宣言	`local val x = 2 in val y = x + x end`

- モジュール言語の宣言

  以下にモジュール宣言 ($\langle moduleDecl \rangle$) の構造と簡単な例を示す.

$$
\begin{array}{lcll}
\langle topdecl \rangle & ::= & \langle strDecl \rangle & \text{ストラクチャ宣言} \\
& | & \langle sigDecl \rangle & \text{シグネチャ宣言} \\
& | & \langle functorDecl \rangle & \text{ファンクタ宣言} \\
& | & \langle localTopdecl \rangle & \text{局所宣言}
\end{array}
$$

ストラクチャ宣言	`structure Version =`
	`  struct`
	`  val version = "3.7.1"`
	`  end`

シグネチャ宣言	`signature VERSION =`
	`  sig`
	`  val version :string`
	`  end`

ファンクタ宣言	`functor System(V:VERSION) =`
	`  struct`
	`  val name = "SML#"`
	`  val version = V.version`
	`  end`

局所宣言	`local`
	`  structure V = Version`
	`in`
	`  structure Release = struct`
	`  val version = V.version`
	`  val date = "2021-03-15"`
	`  end`
	`end`

16.1.1 核言語の宣言の評価

対話型プログラムの実行は，宣言列を順次評価することによって行われる．核言語の宣言の評価は，宣言に含まれる式などの要素を評価し，値を生成し，宣言で定義される名前をそれら値に束縛する効果を持つ．生成される値には，静的な値と，実行時に生成される動的な値がある．

静的な値は，コンパイラがコンパイル時に作り出す値であり，以下の種類がある．

中置演算子属性 識別子が，中置演算子として構文解析されることを表す．結合の向き（右結合，左結合）と強さ（0から9）をもつ．

コンストラクタ属性 識別子がデータコンストラクタであることを表す．コンストラクタ属性を持つ識別子は，パターンマッチングにおいて，対応するコンストラクタとのみマッチする．

型構成子 型構成子は，datatype 宣言で生成される新しい型の定義である．型引数を持つパラメタ型を定義できる．

型 型は，対応するプログラム部分が実行時に生成する動的な値の性質を表現する．

コンパイラは，コンパイル時に宣言に含まれる構成要素が持つ静的な値を生成し，これら静的な値を使って型整合性をチェックしたのち，宣言で定義される識別子を，対応する静的な値に束縛する．さらに，⟨*valDecl*⟩ など実行を伴うプログラムに対して，実行コードを生成する．生成された実行コードは，宣言に含まれる構成要素に対する動的な値を生成し，定義される識別子をそれら動的な値に束縛する．

動的な値には以下の種類がある．

組み込みデータ 第 18 章で定義する組み込み型の実行時表現である．原子型データ，リスト，配列，ベクタを含む．例えば，`int32` は，計算機アーキテクチャが定める 32 ビット符号付き整数データである．

関数クロージャ 関数型の実行時表現である．

データ構成子 データ構造を構成する組み込み関数である．

例外構成子 例外表現を構成する組み込み関数である．

レコード レコードやタプル型の実行時表現である．

データ型表現 データ構成子によって生成されるユーザ定義の datatype 型のデータである．

例外データ 例外構成子で生成される例外を表す実行時表現である．

各宣言が生成する静的な値と動的な値のサマリを以下に示す．

宣言クラス	生成される静的値	生成される動的値	束縛対象
infix 宣言	演算子属性		変数，コンストラクタ名
val 宣言	型	型に応じた動的な値	変数
val rec 宣言	型	関数クロージャ	変数
関数宣言	型	関数クロージャ	変数
データ型宣言	型構成子 コンストラクタ属性	 データ構成子	型構成子名 データ構成子名
型の別名宣言	型関数		型構成子名
例外宣言	コンストラクタ属性	例外構成子	例外構成子名
局所宣言	内容に依存	内容に依存	内容に依存

コンパイラは，静的な値の束縛の集合を表現する静的な型環境 Γ を状態として持ち，実行コードは，動的な値の束縛を表現する環境 E を状態として持つ．宣言の評価はこの環境の下で行われる．式などに含まれる名前は，その名前が現在の環境の中で束縛されている値に評価される．コンパイラによる宣言の評価は，コンパイラの現在の型環境を宣言が生成する静的な束縛で拡張する効果をもつ．宣言の動的な評価，すなわちコンパラが宣言に対して生成したコードの実行は，現在の動的な環境を，宣言が生成する動的な束縛で拡張する効果をもつ．

実行結果の表示等は，各宣言に対する動的な値の生成に伴う副作用によって行われる．さらに，対話型モードでは，静的な値と動的な値を表示する特殊なコードがコンパイラによって付加され，ユーザに表示される．

各宣言の概要と，対話型モードでの評価の例を示す．それぞれの宣言の構文の詳細定義は第 23 章で与える．

⟨*infixDecl*⟩ 指定された識別子 ⟨*id*⟩（のリスト）に演算子属性（インフィックス属性）を与える宣言である．この宣言のスコープでは，⟨*id*⟩ は，中置演算子として使用される．

```
# 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
```

*と^がそれぞれ結合力 7 と 8 の演算子属性を持つので，`4 * 3 ^ 2` は，`*(4, ^ (3,2))` と展開される．この宣言で動的な値は生成されない．

⟨*valDecl*⟩ `val` ⟨*pat*⟩ `=` ⟨*exp*⟩ の形の宣言である．式 ⟨*exp*⟩ を評価えられた値（型および動的な値）とパターン ⟨*pat*⟩ とのパターンマッチングを行い，マッチした値を識別子に束縛する．

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

この例では，変数 `x` に，式 `1` を評価した結果の型 `int` と値 `1` が束縛される．変数以外にも種々の構造を表すパターンを記述できる．

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

式 ⟨*exp*⟩ の詳細は第 19 章で, パターンの詳細は第 20 章で定義する．

⟨*valRecDecl*⟩ 一般に相互再帰的な関数定義に限定した val 宣言である．

```
# 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
```

この宣言により，各識別子が対応する関数型および関数の値に束縛される．

⟨*funDecl*⟩ 一般に相互再帰的な関数定義宣言である．

```
# 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
```

この宣言により，各識別子が対応する関数型および関数の値に束縛される．

⟨*datatypeDecl*⟩ 相互再帰的な新しい型構成子を定義する．

```
# 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
```

この宣言が評価されると，新しい型（パラメタを持たない型構成子）foo 及び bar が作られ，識別子 foo および bar がそれぞれ新しい型構成子に束縛される．さらに，識別子 A, B および C, D にはデータ構成子属性が与えられ，れぞれ foo および bar 型の値を作るデータ構成子に束縛される．

本文書では，特に混乱を引き起こさない限り上記の説明のように，型構成子やデータ構成子を，それらが束縛される名前と同一視する．

⟨*typeDecl*⟩ 識別子を型関数に束縛する宣言である．

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

この宣言により，'a で表される型を受け取って'a * 'a を返す型関数が生成され，識別子 foo に束縛される．これ以降 τ foo は，τ * τ の別名として使用される．

⟨*exceptionDecl*⟩ 例外構成子の宣言である．

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

この宣言が評価されると，型 int を引数とする新しい例外構成子が作られる．この宣言のスコープでは，識別子 Foo にコンストラクタ属性が与えられ，例外構成子に束縛される．

⟨*localDecl*⟩ local および in で囲われた宣言は end まででのみ有効な局所的な宣言である．

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

宣言 val x = 2 のスコープは end までである．従って x はこの局所宣言の外では可視ではなく，対話セッションでも y の値のみが表示される．

16.1.2 モジュール言語の宣言の評価

モジュール言語の構成要素であるストラクチャ宣言は，宣言のリストをひとまとまりにして名前をつける機構である．ストラクチャ式が評価されると，名前の静的な束縛の集合である型環境と動的な束縛の集合である実行時環境が生成される．ストラクチャ宣言は，これら環境で束縛されるそれぞれの名前をストラクチャ名でプリフィックスして得られる型環境と実行時環境を，現在の型環境と実行時環境に追加する効果を持つ．

例えば，ストラクチャ式

```
struct
  val version = "3.7.1"
end
```

は，型環境 {version : string} と実行時環境 {version : "3.7.1"} を生成するので，ストラクチャ宣言

```
structure Version =
  struct
    val version = "3.7.1"
  end
```

は，現在の型環境と実行時環境にそれぞれ，ロング名束縛 {Version.version : string} と {Version.version : "3.7.1"} を追加する効果を持つ．

ストラクチャ式にはシグネチャ制約を付加することができる．シグネチャ制約は，ストラクチャが生成する型環境の静的制約である．変数の値の型の制約以外に，型宣言の制約なども指定できる．シグネチャ制約を持つストラクチャは，シグネチャ制約に指定された名前のみをもつ環境が生成される．

ファンクタは，ストラクチャを受け取りストラクチャを返す関数である．ただし，核言語の関数と異なり，ファンクタ定義はトップレベルに制約され，かつ，ファンクタを受け取るファンクタ等は定義できない．

各宣言の対話型モードでの評価の例を示す．それぞれの宣言の構文の詳細定義は第 24 章で与える．

⟨*strDecl*⟩ ストラクチャを定義する宣言である．

```
# structure Version =
>   struct
>     val version = "3.7.1"
>   end;
structure Version =
struct
  val version = "3.7.1" : string
end
```

この宣言により，識別子 Version がストラクチャを表す型環境に束縛され，ロング識別子 Version.version が，string 型と動的値"3.7.1"に束縛される．

⟨*sigDecl*⟩ 識別子をシグネチャに束縛する宣言である．

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

この宣言により，VERSION がシグネチャに束縛される．シグネチャは，ストラクチャで定義される各名前が持つ型などの属性を表現である．

⟨*functorDecl*⟩ ストラクチャからストラクチャを生成する関数であるファンクタを定義する宣言である．

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

この宣言によって，System が，VERSION シグネチャをもつストラクチャをうけとり，name と version を要素とするストラクチャを返すファンクタに束縛される．

⟨*localTopdecl*⟩ local および in で囲われた宣言は end まででのみ有効な局所的な宣言である．

```
# local
>   structure V = Version
> in
>   structure Release = struct
>     val date = "2021-03-15"
>     val version = V.version
>   end
> end;
val Release =
struct
  val date = "2021-03-15" : string
  val version = "3.7.1" : string
end
```

宣言 structure V = Version のスコープは end までである．従って V はこの局所宣言の外では可視ではなく，対話セッションでも Release のみが表示される．

16.2 分割コンパイルモードのプログラム

分割コンパイルモードのプログラムは，ソースファイルとそのインターフェイスを記述したインターフェイスファイルの集合である．ソースファイルの内容 ⟨*source*⟩ は以下の通りである．

⟨*source*⟩ ::= (⟨*interfaceFileSpec*⟩)? ⟨*sourceProgram*⟩
⟨*interfaceFileSpec*⟩ ::= _interface " ⟨*filePath*⟩ "
⟨*sourceProgram*⟩ ::=
| ⟨*decl*⟩ ⟨*sourceProgram*⟩
| ⟨*topdecl*⟩ ⟨*sourceProgram*⟩

ソースファイルに対応するインタフェイスファイルの指定 ⟨*interfaceFileSpec*⟩ 指定しない場合でかつ，ソースファイル S.sml と同一ディレクトリにサフィクスが.smi の同一の名のファイル S.smi があれば，そのファイルが暗黙にインタフェイスファイルとして指定される．⟨*filePath*⟩ は相対パスであり，OS によらず，ディレクトリの区切り文字には/（スラッシュ）を用いる．

インターフェイスファイルの内容 ⟨*interface*⟩ は，ソースファイルが必要とする他のコンパイル単位の参照宣言の列 ⟨*requireList*⟩ と，ソースファイルが他のコンパイル単位に提供する仕様の列 ⟨*provideList*⟩ を記述する．

⟨*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*⟩ は，ソースファイルが参照する他のソースファイルのインターフェイスファイル（smi ファイル）パス，ライブラリの smi ファイル名，またはシグネチャファイルのパス名を指定する．`_require` に続く `local` は，その`_require` で指定されたインターフェースファイルがソースファイルからのみ参照され，⟨*provideList*⟩ からは参照されないことを表す．この `local` を適切に加えることによって，不要なインターフェースファイルの読み込みを抑止し，コンパイル時間を短縮することができる．`init` は，指定されたインターフェースファイルに対応するプログラムが，`_require` が書かれたプログラムよりも先に必ず評価されることを表す注釈である．⟨*provideList*⟩ は，ソースファイルが，他のコンパイル単位に提供する宣言の仕様である．

⟨*smiFilePath*⟩，⟨*sigFilePath*⟩，および ⟨*librarySmiFilePath*⟩ は，`_require` が書かれたファイルから他のファイルへの相対パスである．OS によらず，ディレクトリの区切り文字には/（スラッシュ）を用いる．パスが．（ピリオド）から始まっておらずかつファイルが存在しない場合は，コマンドラインで与えられたロードパスからファイルを検索する．

SML#が提供する代表的なライブラリのインターフェイスファイルには以下のものがある．

ライブラリ smi ファイル名	内容
basis.smi	Standard ML 基本ライブラリ
ml-yacc-lib.smi	yacc, lex ツール
smlformat-lib.smi	SMLFormat フォーマッタ生成ツール
smlnj-lib.smi	Standard ML of New Jersey ライブラリ
ffi.smi	C との連携サポートライブラリ
thread.smi	マルチスレッドサポートライブラリ
reify.smi	動的型付けサポートライブラリ
smlunit-lib.smi	単体テストツール

宣言に対応するプロバイドは以下の通りである．

- 核言語の宣言とインタフェイスの対応

宣言 (⟨*decl*⟩)	対応するプロバイド (⟨*provide*⟩)
⟨*infixDecl*⟩	⟨*provideInfix*⟩
⟨*valDecl*⟩	⟨*provideVal*⟩
⟨*valRecDecl*⟩	⟨*provideVal*⟩
⟨*funDecl*⟩	⟨*provideVal*⟩
⟨*datatypeDecl*⟩	⟨*provideData*⟩
⟨*typeDecl*⟩	⟨*provideType*⟩
⟨*exceptionDecl*⟩	⟨*provideException*⟩
⟨*localDecl*⟩	

- モジュール言語の宣言とインタフェイスの対応

宣言 (⟨*topdecl*⟩)	対応するプロバイド (⟨*provide*⟩)
⟨*strDecl*⟩	⟨*provideStr*⟩
⟨*sigDecl*⟩	
⟨*functorDecl*⟩	⟨*provideFun*⟩
⟨*localTopdecl*⟩	

シグネチャファイルは，インタフェイスに直接`_require` ⟨*sigFilePath*⟩ 宣言で参照する．

以下に簡単なプログラム例を示す．

- 例 1 （基本ライブラリの使用）

ファイル	コード
`hello.sml`	`val _ = print "こんにちは，SML #へようこそ\n"`
`hello.smi`	`_require "basis.smi"`

コンパイルと実行例

```
$ smlsharp hello.sml
$ ./a.out
こんにちは，SML #へようこそ
$
```

- 例 2 （分割コンパイル）

ファイル	コード
`hello.sml`	`val _ = puts "こんにちは，SML #へようこそ"`
`hello.smi`	`_require "puts.smi"`
`puts.sml`	`val puts = _import "puts" : string -> int`
`puts.smi`	`val puts : string -> int`

コンパイルと実行例

```
$ smlsharp -c hello.sml
$ smlsharp -c puts.sml
$ smlsharp -o hello hello.smi
$ ./hello
こんにちは，SML #へようこそ
$
```

16.3　プログラムの主な構成要素

これら宣言とインタフェイスのそれぞれの構文と意味を定義する前に，まず，これら宣言とインターフェイスを構成する以下の要素の構文と意味を定義する．

字句構造 プログラムを構成する最小単位である予約語や識別子，区切り記号の定義である．(第 17 章)

式 字句を用いてつくられるプログラムの主な構成要素．静的な型を持ち動的な値を生成するプログラムの実行単位である．(第 19 章)

型 式や識別子の持つべき型の指定やインターフェイスの定義に用いられる．(第 18 章)

パターン 関数の仮引数や値の束縛構文などの値の受け取りにおける受け取るべき値の構造の記述である．(第 20 章)

識別子のスコープ規則 識別子の型や動的な値束縛の有効範囲の定義．(第 21 章)

SQL 式 SML#にシームレスに組み込まれた SQL コマンドを表現する式．他の式と同様に多相型を持ち，他の式と混在して使用できる．(第 22 章)

それに続き，第 23 章で核言語の宣言の構文と意味を定義し，第 24 章でモジュール言語の宣言の構文と意味を定義する．

第17章　字句構造

本章では，SML#言語で使用する文字と字句構造を定義する．字句構造の定義では，標準の正規表現を用いる．

17.1　文字集合

SML#言語で使用できる文字は，十進数で 0 から 255 までの拡張 ASCII 文字である．文字の中で，十進数で 0 から 127 までの ASCII 文字は，SML#言語の予約語や区切り記号として使用される．これら予約語や区切り記号と重ならない限り，十進数で 128 から 255 までの 8 ビットバイトを含む文字を識別子として使用できる．この機能により，UTF-8 でエンコードされた日本語を識別子とし使用できる．以下の対話型セッションは，日本語を使用したプログラム例である．

```
$ smlsharp
SML# 3.7.1 ...
# 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
```

日本語の Shift_JIS コードなどのような，十進数 0 から 127 までのバイトが現れるエンコーディングを使用した場合の動作は未定義である．

17.2　字句集合

SML#の字句は，以下の種類がある．

予約語 以下は SML#の予約語であり，識別子には使用できない．

```
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 ( ) [ ] { } , : :> ; ... _ = => -> #
```

これら文字列は，以下の識別子の項で定義される言語クラス $\langle alphaId \rangle$ には含まれない．

SQL 予約語 以下の各語は，SQL 構文の中では予約語であり，識別名として使用できない．

```
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
```

これら文字列は，第 19 章および第 22 章で定義される，`_sql` で始まる SQL 式内では，字句クラス ⟨*alphaId*⟩ には含まれない．その詳細は第 22 節で述べる．SQL 構文は，Standard ML では構文として認識されないため，これら制限を追加しても Standard ML との後方互換性が保たれている．

拡張予約語 `_`で始まる以下の名前は，SML#の拡張機能を実現するために導入された予約語である．これら拡張予約語は，Standard ML では字句と認識されない語であり，これらを追加しても Standard ML との後方互換性が保たれている．

```
__attribute__ _builtin _foreach _import _interface _join _dynamic _dynamiccase
_polyrec _require _sizeof _sql _sqlserver _typeof _use
```

識別子 識別子は，プログラム中で使用される種々の名前である．SML#では，以下の 7 種類の識別子がある．これら 7 種類はその使用文脈に応じて区別されるので，同一の名前を別の用途に使用してよい．

名	用途
⟨*vid*⟩	変数，データコンストラクタ
⟨*lab*⟩	レコードのラベル
⟨*strid*⟩	ストラクチャ名
⟨*sigid*⟩	シグネチャ名
⟨*funid*⟩	ファンクタ名
⟨*tycon*⟩	型構成子名
⟨*tyvar*⟩	型変数

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*⟩ （注 1）
⟨*strid*⟩ ::= ⟨*alphaId*⟩
⟨*sigid*⟩ ::= ⟨*alphaId*⟩
⟨*funid*⟩ ::= ⟨*alphaId*⟩
⟨*tyvar*⟩ ::= ' ⟨*alphaId*⟩ | '' ⟨*alphaId*⟩
⟨*tycon*⟩ ::= ⟨*alphaId*⟩ | ⟨*symbolId*⟩

注

1. レコードラベルは，文字列ラベル（⟨*alphaId*⟩ または ⟨*string*⟩），整数ラベル（⟨*decimal*⟩），順序付き文字列ラベル（⟨*decimal*⟩ _ ⟨*alphaId*⟩）の 3 種類がある．SML#では，レコードは，ラベルの順序に従ってソートされ，メモリーが割り当てられる．文字列ラベルの順序は，文字列の順序 `String.compare` である．数字ラベルの順序は，数字列を自然数と解釈した時の自然数の順序である．順序付き文字列ラベルは，数字ラベルと文字列ラベルが_で連結されたものであり，その順序は，数字ラベルと文字列ラベルの順序の辞書式順序である．

各文字クラスの定義は以下の通り．

⟨*alpha*⟩	::=	[A-Za-z\127-\255]	
⟨*symbol*⟩	::=	! | % | & | $ | + | / | : | < | = | > | ? | @ | ~ | ‘ | | | # | - | ^ | \	
⟨*alphaId*⟩	::=	⟨*alpha*⟩ (⟨*alpha*⟩ | [0-9] | ’ | _)*	(注 1)
⟨*decimal*⟩	::=	[1-9] [0-9]*	
⟨*symbolId*⟩	::=	⟨*symbol*⟩ *	(注 2)

注

1. 字句クラス ⟨*alphaId*⟩ には予約語は含まれない．さらに，_sql で始まる SQL 式内では，⟨*alphaId*⟩ には SQL 予約語は含まれない．
2. ⟨*symbolId*⟩ には予約語は含まれない．したがって，==>は ⟨*symbolId*⟩ の要素であるが，=>は ⟨*symbolId*⟩ の要素ではない．

long 識別子 変数やデータ構成子名 ⟨*vid*⟩ とストラクチャ名 ⟨*strid*⟩ に対しては，ストラクチャ名のリストがプレフィックスされた long 識別子が定義される．

⟨*longVid*⟩	::=	(⟨*strid*⟩ .)* ⟨*vid*⟩
⟨*longTycon*⟩	::=	(⟨*strid*⟩ .)* ⟨*tycon*⟩
⟨*longStrid*⟩	::=	(⟨*strid*⟩ .)* ⟨*strid*⟩

定数リテラル ⟨*scon*⟩ 定数リテラルの定義は以下の通りである．

⟨*scon*⟩	::=	⟨*int*⟩ | ⟨*word*⟩ | ⟨*real*⟩ | ⟨*string*⟩ | ⟨*char*⟩	定数リテラル
⟨*int*⟩	::=	(~)?[0-9]+	10 進整数
	|	(~)?0x[0-9a-fA-F]+	16 進整数
⟨*word*⟩	::=	0w[0-9]+	10 進符合なし整数
	|	0wx[0-9a-fA-F]+	16 進符合なし整数
⟨*real*⟩	::=	(~)?[0-9]+ . [0-9]+ [Ee](~)?[0-9]+	不動小数点数
	|	(~)?[0-9]+ . [0-9]+	
	|	(~)?[0-9]+ [Ee](~)?[0-9]+	
⟨*char*⟩	::=	#" (⟨*printable*⟩ | ⟨*escape*⟩) "	文字
⟨*string*⟩	::=	" (⟨*printable*⟩ | ⟨*escape*⟩)* "	文字列
⟨*printable*⟩	::=	\と"を除いた文字	
⟨*escape*⟩	::=	\a	警告文字 (ASCII 7)
	|	\b	バックスペース (ASCII 8)
	|	\t	タブ (ASCII 9)
	|	\n	改行文字 (ASCII 10)
	|	\v	垂直タブ (ASCII 11)
	|	\f	ホームフィード (ASCII 12)
	|	\r	リターン (ASCII 13)
	|	\^[\064-\095]	[\064-\095] が表す制御文字
	|	\\	文字\
	|	\"	文字"
	|	*ddd*	十進数 *ddd* の番号を持つ文字
	|	\$f \cdots f$\	空白文字の列 $f \cdots f$ を無視する
	|	\u*xxxx*	16 進数表現 *xxxx* で表される番号を持つ UTF

第18章　型

本節では，型の構文およびSML#コンパイラに組み込まれている型を定義する．
型は，単相型 (⟨*ty*⟩) と多相型 (⟨*polyTy*⟩) に分類される．単相型の集合は，以下の文法であたえあれる．

⟨*ty*⟩	::=	⟨*tyvar*⟩	型変数名
	|	`{`(⟨*tyrow*⟩)?`}`	レコード型
	|	⟨ty_1⟩ `*` $\cdots$ `*` ⟨ty_n⟩	組型 ($n \geq 2$)
	|	⟨*ty*⟩ `->` ⟨*ty*⟩	関数型
	|	(⟨*tySeq*⟩)? ⟨*longTycon*⟩	(パラメタ付) きデータ型
	|	`(` ⟨*ty*⟩ `)`	
⟨*tyrow*⟩	::=	⟨*lab*⟩ `:` ⟨*ty*⟩ (`,` ⟨*tyrow*⟩)?	レコードフィールドの型

⟨*tyvar*⟩ は，型の集合を動く変数である．第 17.2 節で定義した通り`'a`，`'foo` および`''a`，`''foo` のように表記される．後者は同値プリミティブ演算が適用可能な eq 型に限定された型変数である．eq 型は，関数型と eq 型ではない組み込み型（`real`, `real32`, `exn`）を含まない任意の型である．ユーザ定義のデータ型も，その定義に含まれる型がすべて eq 型であれば eq 型である．従って，例えば以下のように定義される τ `list` は引数の型 τ が eq 型であれば eq 型である．

```
datatype 'a list = nil | :: of 'a * 'a list
```

関数型構成子`->`はラムダ計算の伝統に従い右結合する．したがって，`int -> int -> int` と記述すると，`int -> (int -> int)` と解釈される．
⟨*longTycon*⟩ は，`datatype` 宣言で定義される型構成子の名前であり，宣言された structure のパス名で修飾される．`int` 型等の基底型はパラメタ ⟨*tySeq*⟩ を持たないデータ型である．SML#3.7.1 には以下の基底型およびデータ型構成子が組み込まれている．

型構成子名	説明	eq 型？
`int`	32 ビット符号付き整数	Yes
`int64`	64 ビット符号付き整数	Yes
`int16`	16 ビット符号付き整数	Yes
`int8`	8 ビット符号付き整数	Yes
`intInf`	桁数制限のない符号付き整数	Yes
`word`	32 ビット符号なし整数	Yes
`word64`	64 ビット符号なし整数	Yes
`word16`	16 ビット符号なし整数	Yes
`word8`	8 ビット符号なし整数	Yes
`real`	浮動小数点数	No
`real32`	32 ビット浮動小数点数	No
`char`	文字	Yes
`string`	文字列	Yes
`exn`	例外	No
`unit`	ユニット値 (`()`)	Yes
τ `ref`	参照（ポインタ）	Yes
τ `array`	配列	Yes
τ `vector`	ベクトル	τ が eq 型ならば Yes

ユニット値の型 `unit` と空のレコード型`{}`は異なる型として区別される．これは Standard ML に対する後方互換性の無い変更点である．

多相型の集合は以下の文法で与えられる．

⟨*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*⟩ `]` は，束縛型変数 ⟨*boundtyvarList*⟩ のスコープが明示された多相型である．
- 束縛型変数は，その取りうる型の集合を制限する以下のカインド制約 ⟨*kind*⟩ をつけることができる．レコードカインド`#{` ⟨*tyrow*⟩ `}`は，⟨*tyrow*⟩ が表すフィールドを含むレコード型に制限する．`boxed` カインドはヒープにアロケートされる値の型に制限する．`unboxed` カインドはヒープにアロケートされない値の型に制限する．`eq` カインドは eq 型に制限する．`reify` カインドは型のリーフィケーション機能を用いることの注釈であり，型変数が動く範囲を制限しない．

この集合は，Standard ML の多相型をレコード多相性およびランク 1 多相に拡張したものである．例えば，以下のような多相型を持つ関数が定義できる．

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

対話セッションでは，上記のカインドに加えて，オーバロードカインド`::{` $\langle tyList \rangle$ `}`がプリントされることがある．これは，型変数が動く範囲をインスタンス型 $\langle tyList \rangle$ のいずれかに制限するカインドである．現在，オーバーローディングはシステム定義のプリミティブのみに限定されており，オーバーロードカインドをもつ多相型をユーザプログラムが指定することはできない．

第19章　式

式 ⟨*exp*⟩ の構文の定義は，演算子式 ⟨*infix*⟩，関数適用式 ⟨*appexp*⟩，原子式 ⟨*atexp*⟩ を用いて階層的に定義される．

- 式（トップレベル）

⟨*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*⟩　C 関数のインポート
| ⟨*exp*⟩ `:` `_import` ⟨*cfunty*⟩　C 関数のインポート
| `_sizeof(` ⟨*ty*⟩ `)`　型のサイズ
| `_dynamic` ⟨*exp*⟩ `as` ⟨*ty*⟩　動的型キャスト
| `_dynamiccase` ⟨*exp*⟩ `of` ⟨*match*⟩　動的型キャスト付き場合分け
| `_sqlserver` (⟨*appexp*⟩)? `:` ⟨*ty*⟩　SQL サーバ
| `_sql` ⟨*pat*⟩ `=>` ⟨*sqlfn*⟩　SQL 実行関数
| `_sql` ⟨*sql*⟩　SQL クエリ断片

⟨*match*⟩ ::= ⟨*pat*⟩ `=>` ⟨*exp*⟩ (`|` ⟨*match*⟩)?　パターンマッチング

- 演算子式

⟨*infix*⟩ ::= ⟨*appexp*⟩
| ⟨*infix*⟩ ⟨*vid*⟩ ⟨*infix*⟩

- 関数適用式

⟨*appexp*⟩ ::= ⟨*atexp*⟩
| ⟨*appexp*⟩ ⟨*atexp*⟩　関数適用（左結合）
| ⟨*appexp*⟩ `#{` ⟨*exprow*⟩ `}`　レコードフィールドアップデート

- 原子式

⟨*atexp*⟩	::=	⟨*scon*⟩	定数
	|	(`op`)? ⟨*longVid*⟩	識別子
	|	`{`(⟨*exprow*⟩)? `}`	レコード
	|	`(`⟨exp_1⟩`,`$\cdots$`,`⟨exp_n⟩`)`	組（$n \geq 2$）
	|	`()`	`unit` 型定数
	|	`#`⟨*lab*⟩	レコードフィールドセレクト
	|	`[`⟨exp_1⟩`,`$\cdots$`,`⟨exp_n⟩`]`	リスト（$n \geq 0$）
	|	`(`⟨exp_1⟩`;`$\cdots$`;`⟨exp_n⟩`)`	逐次実行式（$n \geq 2$）
	|	`let` ⟨*declList*⟩ `in` ⟨exp_1⟩`;`$\cdots$`;`⟨exp_n⟩ `end`	局所宣言（$n \geq 1$）
	|	`_sql (`⟨*sql*⟩`)`	SQL クエリ断片
	|	`(`⟨*exp*⟩`)`	
⟨*exprow*⟩	::=	⟨*lab*⟩ `=` ⟨*exp*⟩ (`,` *exprow*)?	レコードフィールド

⟨*cfunty*⟩ の定義は第 19.21 節に，⟨*sql*⟩ および ⟨*sqlfn*⟩ の定義は第 22.2 節にそれぞれ与える．

この階層によって，構文の結合力（まとまりの強さ）が表現される．原子式が結合の最小単位であり，関数適用式 ⟨*appexp*⟩，演算子式 ⟨*infix*⟩，式 ⟨*exp*⟩ の各定義は，この順の結合力が強い．ただし，演算子式 ⟨*infix*⟩ を構成する各要素間の結合力は，文法ではなく，演算子宣言によって決定される．さらに演算子式は，演算子宣言の結合の強さに従って，組への関数適用に展開される糖衣構文である．そこでまず，次節 (19.1) において演算子式の展開を定義したのち，式の文法で表現される結合力の強順に，各構造とその式がもつ型と値を説明する．

19.1 演算子式の展開

演算子宣言は，構文構造の決定に関わる特殊宣言であり，以下の構文で定義される．

`infix` (`n`)? ⟨*vidList*⟩
`infixr` (`n`)? ⟨*vidList*⟩

これら宣言は，⟨*vid*⟩ に対して演算子属性を与える．`infix` 宣言は，識別子が左結合する演算子として使用することを宣言し，`infixr` 宣言は，識別子が右結合する演算子として使用することを宣言する．オプションとして結合力を 0 から 9 までの数字 ⟨*n*⟩ で指定することができる．数字が大きいほうが結合力が強い．結合力 n を省略すると，0 を指定したとみなされる．宣言，

`nonfix` ⟨*vidList*⟩

は，⟨*vidList*⟩ の演算子属性を削除する．

これら演算子宣言は，⟨*vid*⟩ の宣言が書ける所に書くことができ，⟨*vid*⟩ の宣言が生成する識別子の定義と同様の静的スコープに従うが，構文解析処理への指示であり，識別子の定義とは独立になされる．

演算子属性が与えられた識別子 ⟨*id*⟩ が式またはパターンに現れると，演算子の優先度に従い以下の構文変換が行われる．

変換前	変換後
⟨exp_1⟩ ⟨*vid*⟩ ⟨exp_2⟩	`op` ⟨*vid*⟩ (⟨exp_1⟩`,`⟨exp_2⟩)
⟨pat_1⟩ ⟨*vid*⟩ ⟨pat_2⟩	`op` ⟨*vid*⟩ (⟨pat_1⟩`,`⟨pat_2⟩)

ここで，⟨exp_1⟩ と ⟨exp_2⟩ は，演算子属性の結合力の強さに応じて決定される．式またはパターンに現れた構文

`op` ⟨*vid*⟩

は，⟨*vid*⟩ と同一であるが，⟨*vid*⟩ の演算子属性は無視される．

従って，`foo` が演算子属性をもつ識別子の場合，以下の 2 つの式は等価である．

```
1 foo 2
op foo (1,2)
```

演算子宣言はインタフェイスファイルでも行うことができる．SML#言語では，すべてのコンパイル単位および対話型モードで，以下の演算子宣言を含むインタフェイスファイルが暗黙に_require されている．

```
infix 7 * / div mod
infix 6 + - ^
infixr 5 :: @
infix 4 = <> > >= < <=
infix 3 := o
infix 0 before
```

以上の階層的な構文の定義とインフィックス宣言によって，構文間の結合の強さが定義される．例えば，レコードアップデート構文 (⟨exp_1⟩ `#` `{`⟨*lab*⟩`=`⟨exp_2⟩`}`) の結合力は，最も結合力の強い（infix 9 と宣言された）演算子よりも強く，`if` ⟨*exp*⟩ `then`⟨*exp*⟩ `else` ⟨*exp*⟩ 等の他の構文の結合力は，最も結合力の弱い（infix 0 と宣言された）演算子の結合力よりも弱い．

19.2 定数式 ⟨*scon*⟩

定数式は，第 17.2 節で定義した定数リテラル ⟨*int*⟩，⟨*word*⟩，⟨*real*⟩，⟨*string*⟩，⟨*char*⟩ である．

これらの中で ⟨*int*⟩，⟨*word*⟩，⟨*real*⟩ の各定数は以下の表の通りオーバロードされており，使用文脈によって適当な型が選ばれる．文脈の制約がなければ，デフォルトの型を持つ．

クラス	取りうる型集合	デフォルトの型
⟨*int*⟩	`int`, `int32`, `int64`, `int8`, `int16`, `intInf`	`int`
⟨*word*⟩	`word`, `word32`, `word64`, `word8`, `word16`	`word`
⟨*real*⟩	`real`, `real32`	`real`

⟨*string*⟩ 及び ⟨*char*⟩ リテラルは，それぞれ `string` 及び `char` 型を持つ，

また，これら定数リテラルは，型に応じてさだまるアーキテクチャ上の自然な動的な値を生成する．実行時の値表現の詳細は第 29 章で定義する．

以下は，定数式の評価の例である．

```
# 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 識別子式 ⟨*longVid*⟩

式として現れた long 識別子 ⟨*longVid*⟩ の型と値は，現在の環境の中で，その ⟨*longVid*⟩ が束縛された型と値である．

⟨*longVid*⟩ は，変数識別子 ⟨*vid*⟩ にストラクチャ識別子 ⟨*strId*⟩ の並びがプレフィックスされた ⟨$strId_1$⟩ . ⋯ . ⟨$strId_n$⟩ . ⟨*vid*⟩ の形の式である．各 ⟨$strId_i$⟩ は，第 21 章で定義する静的スコープ規則によって，ネストしたストラクチャ宣言に対応する．ストラクチャ宣言は，第 16 章で定義されるように，現在の環境にロング識別子の束縛の集合追加する効果を持つ．さらに，⟨$strId_{i-1}$⟩ が生成するロング識別子の束縛の集合は，⟨$strId_i$⟩ が生成するロング識別子の束縛の集合にストラクチャ名 ⟨*strId*⟩ を付加して得られるロング識別子の束縛の集合を含む．

従って，⟨*longVid*⟩ の型と値は，⟨$strId_n$⟩ に対応するストラクチャの中で ⟨*vid*⟩ に束縛された型と値と同じである．n がゼロの場合は，トップレベルで ⟨*vid*⟩ に束縛された型と値である．簡単な例を以下に示す．

```
# 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 レコード式 { ⟨lab_1⟩ = ⟨exp_1⟩ , . . . , ⟨lab_n⟩ = ⟨exp_n⟩ }

レコード式はラベル ⟨lab_i⟩ と式 ⟨exp_i⟩ の組みからなるレコードフィールドの集合である．
レコード式は以下のように評価される．

1. レコード式のラベルがすべて異なるかチェックされ，同一のラベルが複数現れれば，型エラーとなる．
2. 次に，各式 ⟨exp_i⟩ が，レコード式に出現する順に評価され，その型 ⟨ty_i⟩ と値 v_i が計算される．
3. それらを使い，ラベル文字列でソートされたコード型

   { ⟨lab'_1⟩ : ⟨ty'_1⟩ , . . . , ⟨lab'_n⟩ : ⟨ty'_n⟩ }

   とレコード値

{⟨lab'_1⟩ = v'_1, ..., ⟨lab'_n⟩ = v'_n}

が求められ，レコード式の型と値となる．

従って，例えば

```
val x = {SML = (print "SML#";"SML#"), IS = (print " is ";" is "), SHARP = (print
" sharp!\n";" sharp!\n")}
```

を評価すると，"SML# is sharp!"とプリントされた後，

```
val x = {IS = " is ", SHARP = " sharp!\n", SML = "SML#"} : {IS: string, SHARP:
string, SML: string}
```

のような型と値が x に束縛される．

19.5 組式 (⟨exp_1⟩ ,···, ⟨exp_n⟩) と unit 式 ()

組式 (⟨exp_1⟩ , ···, ⟨exp_n⟩) は，数字をラベルとするレコード式{1 = ⟨exp_1⟩ , ···, n = ⟨exp_n⟩} の糖衣構文である．構文的にレコード式に変換された後に，レコードとして評価される．ただし，SML#の対話型コンパイラは，数字をラベルとするレコードの型と値を組として表記する．

unit 型は空の組型である．SML#では，Standard ML の定義とは異なり，unit 型と空のレコード型{} は異なる型として扱われる．空のレコード型{}は空のレコードカインドを持つ．

以下は，対話型環境での組式の評価の例である．

```
# 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 フィールドセレクタ式 #⟨lab⟩

以下の型をもつ多相型フィールドセレクタプリミティブ関数である．

```
['a#{⟨lab⟩:'b}, 'b. 'a -> 'b]
```

この型は，'b の ⟨*lab*⟩ フィールドを含むレコード型'a から型'b への多相関数を表す．この型から，このプリミティブが適用可能な式は，ラベル ⟨*lab*⟩ を含む種々のレコード型を持つものに制約される．このプリミティブの適用結果の型と値は，適用された ⟨*lab*⟩ を含むレコードの ⟨*lab*⟩ フィールドの型と値である．

以下は，レコードフィールドセレクタを含む式の評価例である．

```
# #y;
val it = fn : ['a#{y: 'b}, 'b. 'a -> 'b]
# #y {x = 1, y = 2};
val it = 2 : int
# #2;
val it = fn : ['a#{2: 'b}, 'b. 'a -> 'b]
# #2 (1, 2, 3);
val it = 2 : int
```

組型は，レコード型の特殊な場合であるため，組みに対してもレコードフィールドセレクタが使用できる．

19.7 リスト式 [⟨exp_1⟩ , ···, ⟨exp_n⟩]

リスト式は，[と] の中にコンマで区切った式の値のリストであり，リスト型のその値を生成する．リスト型は，予め以下のように定義されたデータ型である．

```
infixr 5 ::
datatype 'a list = op :: of 'a * 'a list | nil
```

リスト式は以下のように構文変換が行われ，その結果のネストしたコンストラクタ適用式が評価される．

変換前	変換後
[⟨exp_1⟩ ,···, ⟨exp_n⟩]	⟨exp_1⟩ :: ··· :: ⟨exp_n⟩ :: nil

$(0 \leq n)$

この変換から，リスト式の要素の式はすべて同じ型を持たなければならず，その型を要素の型とするリスト型 ty list が，この式の型となり，::(v_1, ::(v_2,··· ::(v_n,nil)···)) の形の値がこの式の値となることがわかる．

以下にリスト式の例を示す．

```
# [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]
```

最後の例のように，[] は多相型を持つ空のリストである．

19.8 逐次実行式 (⟨exp_1⟩ ; ···; ⟨exp_n⟩)

⟨exp_1⟩ から ⟨exp_n⟩ までの式を，この順に実行し，最後の式の型と値を返す式である．SML#の式の評価は一般に副作用を含む．この逐次実行式は，主に，副作用を制御するために用いられる．以下に使用例を示す．

```
# 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，:=，!は，第 19.20 節で説明する参照型（ポインター型）を操作する組み込み演算である．

19.9 局所宣言式 let ⟨declList⟩ in ⟨exp$_1$⟩ ;···; ⟨exp$_n$⟩ end

式 let ⟨declList⟩ in ⟨exp$_1$⟩ ;···; ⟨exp$_n$⟩ end によって，式 ⟨exp$_1$⟩ ;···; ⟨exp$_n$⟩ だけで有効な種々の宣言を行うことができる．この式は以下のように評価される．

1. ⟨declList⟩ が順に評価され，その結果の変数束縛の環境が現在の環境に追加される．
2. 追加された環境の下で式 ⟨exp$_1$⟩ ;···; ⟨exp$_n$⟩ が，この順に評価される．
3. 最後に評価された式の型と値が，この式全体の型と値となる．

19.10 関数適用式 ⟨appexp⟩ ⟨atexp⟩

以上に定義した原子式は，構文自体に式の区切りを含んいる式の最小単位である．式はこの原子式を式構成子によって組み合わせて構成されている．組み合わせの中で最も結合力が強い式構成子が，関数適用である．ラムダ式の伝統を引き継ぐ SML#では，関数適用は，文法 ⟨appexp⟩ ⟨atexp⟩ に示すように，関数を表す式 ⟨appexp⟩ と引数を表す式 ⟨atexp⟩ を単に式を並べて記述する．⟨exp1⟩ ⟨exp1⟩ ⟨exp3⟩ のように連続して並べられた式は，この文法から，左結合する関数適用のネスト ((⟨exp1⟩ ⟨exp1⟩) ⟨exp3⟩) と解釈される．

関数適用式 ⟨appexp⟩ ⟨atexp⟩ の評価は以下のように行われる．

1. 式 ⟨appexp⟩ を静的に評価し，型を求める．もし型が関数型 ⟨ty$_1$⟩ -> ⟨ty$_2$⟩ でなければ，型エラーとなる．
2. 式 ⟨atexp⟩ を静的に評価し，⟨ty$_3$⟩ 型を求める．⟨ty$_1$⟩ と ⟨ty$_3$⟩ が同一でなければ型エラーとなる．
3. 式 ⟨appexp⟩ を動的に評価し，値 *Cls* を求める．値は関数クロージャである．
4. 式 ⟨atexp⟩ を動的に評価し，値 v を求める．関数クロージャ *Cls* に保存された環境に，仮引数の値 v への束縛を追加し，その環境の下で，関数クロージャ *Cls* に保存されたコードを実行し，動的な値 v' を求める．型 ⟨ty$_2$⟩ と v' が，関数適用式の型と値である．

以下に関数適用式を含む例を示す．最後の例では，#name f("joe", 21) は ((#name f) ("joe", 21)) と解釈され，型エラーとなる．

```
# 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 フィールドアップデート式 ⟨*appexp*⟩ # { ⟨*exprow*⟩ }

式 ⟨*appexp*⟩ の値がレコードの時，そのフィールドを{ ⟨*exprow*⟩ }で指定した値に変更して得られる新しいレコードを生成する式である．この式の型は，式 ⟨*appexp*⟩ の型と同一である．以下に例を示す．

```
# {x = 1, y = 2} # {x = 2};
val it = {x = 2, y = 2} : {x : int, y : int}
```

以下の例が示す通り，この式はレコードに関して多相型を持つ．

```
# 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 型制約式 ⟨*exp*⟩ : ⟨*ty*⟩

式 ⟨*exp*⟩ の型を ⟨*ty*⟩ に制約する式である．⟨*exp*⟩ は型 ⟨*ty*⟩ より一般的な型である必要がある．以下に例を示す．

```
# [] : 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]
```

最後の例のように ⟨*ty*⟩ は型変数を含んでも良い．含まれる型変数は，インスタンス化されない．また型変数のスコープは，特に型変数の宣言がなければ，その型変数が現れるもっとも内側の val 宣言全体である．型変数の宣言およびそのスコープ規則については，第 23.1 節で定義する．

19.13 論理演算式 ⟨exp_1⟩ andalso ⟨exp_2⟩ および ⟨exp_1⟩ orelse ⟨exp_2⟩

真理値型は以下のように予め定義されている．

```
datatype bool = false | true
```

通常データ構成子は大文字で始まる習慣があるが，false と true は例外である．

bool 型を持つ 2 つの式 ⟨exp_1⟩ と ⟨exp_2⟩ に対して以下の構文が定義されている．

- 論理積：⟨exp_1⟩ andalso ⟨exp_2⟩

  ⟨exp_1⟩ の値を評価し，その結果が true であれば，⟨exp_2⟩ を評価し，その結果が式全体の結果となる．⟨exp_1⟩ の値が false であれば，false が式の値である．この場合，⟨exp_2⟩ は評価されない．

- 論理和：⟨exp_1⟩ orelse ⟨exp_2⟩

  ⟨exp_1⟩ の値を評価し，その結果が false であれば，⟨exp_2⟩ を評価し，その結果が式全体の結果となる．⟨exp_1⟩ の値が true であれば，true が式の値である．この場合，⟨exp_2⟩ は評価されない．

このような評価を実現するために，これら演算は関数ではなく構文として提供されている．

これら構文はいずれも左結合する．また，同一レベルの構文定義では，先に現れた構文の結合力は後に出現した構文より強い．従って，

```
false andalso true orelse false
```

は (false andalso true) orelse false と解釈され，評価結果の値は false となる．

bool 型の演算には，これら構文以外に関数が定義されている．

```
not : bool -> bool
```

19.14 例外処理式 ⟨*exp*⟩ handle ⟨*match*⟩

式 ⟨*exp*⟩ 評価中に発生する例外をキャッチし処理する構文である．⟨*match*⟩ では，例外パターンとそのパターンにマッチした例外の時評価される式の組を関数式と同様に以下の形で記述する．

⟨pat_1⟩ => ⟨exp_1⟩
| ⋯
| ⟨pat_n⟩ => ⟨exp_n⟩

各パターン ⟨pat_i⟩ は例外構成子を含むパターンであり例外型 (exn) を持たねばならない．また，各式 ⟨*exp*⟩$_i$ の型は ⟨*exp*⟩ の型と同一である必要がある．この式の評価は以下のように行われる．

- 式 ⟨*exp*⟩ を評価する．評価が正常に終了すれば，その値がこの式全体の値となる．
- 式 ⟨*exp*⟩ 中に例外が発生すれば，その例外がパターン ⟨pat_1⟩ から ⟨pat_n⟩ とこの順に照合され，マッチするパターン ⟨pat_i⟩ があれば，⟨pat_i⟩ の中の変数が対応する例外パラメータに束縛され，その束縛が追加された環境で，式 ⟨exp_i⟩ を評価し，その結果が式全体の結果となる．
- マッチする例外パターンがなければ，この式で同一の例外が発生したもとして，この式を囲む式の評価が続行される．

19.15 例外発生式 raise ⟨*exp*⟩

例外型（exn）を持つ任意の ⟨*exp*⟩ に対して，この式の型は，文脈の制約によって決まる任意の型を持つ．文脈の制約がなければ型変数'a で表現される多相型を持つ．⟨*exp*⟩ が例外型を持たなければ，この式は型エラーとなる．

この式の評価を評価すると，まず式 ⟨*exp*⟩ が評価され，その結果の例外値を持つ例外が発生した状況が生成される．したがって，この式は値を持たない．

19.16 条件式 if ⟨exp_1⟩ then ⟨exp_2⟩ else ⟨exp_3⟩

条件分岐を表す式である．式 ⟨exp_1⟩ は bool 型ち，式 ⟨exp_2⟩ と式 ⟨exp_3⟩ は同じ型を持つことが要求される．この制約の下で，この式は ⟨exp_2⟩ と同じ型を持つ．

この式の評価は以下のように行われる．⟨exp_1⟩ を評価し，その値が true であれば式 ⟨exp_2⟩ を評価しその値をこの式の評価の値とする．⟨exp_1⟩ の値が false であれば式 ⟨exp_3⟩ を評価しその値をこの式の評価の値とする．

19.17 while 式 while ⟨exp_1⟩ do ⟨exp_2⟩

⟨exp_1⟩ が bool 型を持つ式であれば，この式全体は unit 型を持つ．この式の評価は，⟨exp_1⟩ の値が true であれば ⟨exp_2⟩ を評価することを繰り返し．⟨exp_1⟩ の値が false であれば，() を返す．

19.18 場合分け式 case ⟨exp⟩ of ⟨$match$⟩

⟨exp⟩ の評価結果の値を場合分けで処理する構文である．⟨$match$⟩ では，データ構成子を含むパターンとそのパターンにマッチした時評価される式の組を以下の形で記述する．

⟨pat_1⟩ => ⟨exp_1⟩
| ⋯
| ⟨pat_n⟩ => ⟨exp_n⟩

各パターン ⟨pat_i⟩ は，第 20 章で定義するデータ構成子と変数からなるデータパターンである．これらパターン集合は以下の制約を満たさなければならない．

- 同一のパターンに含まれる現れる変数はすべて異なる．
- パターンはすべて ⟨exp⟩ の型と同じ型を持つ．
- $2 \leq i \leq n$ なるすべての i について，パターン ⟨pat_i⟩ が冗長であってはならない．すなわち ⟨pat_i⟩ がカバーするデータの集合が，パターン ⟨pat_1⟩ から ⟨pat_{i-1}⟩ がカバーする集合の和に完全に含まれてはならない．

例えば以下の例は 3 番目の条件に違反している．

```
# 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) => ...
```

この制約のもとで，この式全体は各式 ⟨exp_i⟩ の型と同じ型を持つ．

この場合分け式の評価は以下のように行われる．⟨exp⟩ を評価し得られた値を，⟨pat_1⟩ から ⟨pat_n⟩ のパターンに対して，この順にマッチングを試み，最初にマッチしたパターン ⟨pat_i⟩ に含まれる変数を対応する値に束縛し，その束縛を現在の環境追加して得られる環境で，⟨pat_i⟩ に対応する式 ⟨exp_i⟩ を評価し，得られる値を，この式の値とする．マッチするパターンが無い場合は Match 例外を発生させる．

19.19 関数式 fn ⟨match⟩

⟨match⟩ が表現する関数型とクロージャを生成する．⟨match⟩ は，データ構成子を含むパターンと，そのパターンにマッチしする引数が与えられた時評価される式の組である．

```
  ⟨pat1⟩ => ⟨exp1⟩
| ···
| ⟨patn⟩ => ⟨expn⟩
```

各パターンと式の組 ⟨pat_i⟩ => ⟨exp_i⟩ に対して，⟨pat_i⟩ の型 ⟨ty_i⟩ を求め，パターンの中の変数の型を現在の型環境に加えた型環境の下で ⟨exp_i⟩ の型 ⟨ty'_i⟩ を求め，関数型 ⟨ty_i⟩ -> ⟨ty'_i⟩ を得る．これら関数型をすべて単一化して得られる関数型 ⟨ty⟩ -> ⟨ty'⟩ がこの関数式の型である．関数式の値は，現在の値の環境とこの関数式の組みからなる関数クロージャである．

19.20 組み込み型とその演算

SML#の組み込み型の中で，例外型 (exn) の操作は，第 19.14 節で定義した handle 構文と raise 構文で行われる．ユニット型 (unit) に対する特別な操作はない．それ以外の組み込み型の値の操作は，組み込み関数として提供され，関数適用式を通じて使用される．

参照型 ('a ref) と配列型 ('a array) 以外の値は，自然な関数型言語の意味論に従う値であり，組み込み関数は値を受け取り値を返す関数である．

参照型 ('a ref) と配列型 ('a array) は，メモリー領域の破壊的な更新を許す．

⟨ty⟩ ref の値は ⟨ty⟩ 型の値へのポインターであり，以下の組み込み関数で操作される．

```
ref : ⟨ty⟩ -> ⟨ty⟩ ref
  ! : ⟨ty⟩ ref -> ⟨ty⟩
 := : ⟨ty⟩ ref * ⟨ty⟩ -> unit
infix 3 :=
```

ref ⟨exp⟩ によって，⟨exp⟩ の評価結果の値を指すポインタが生成される．ref ⟨exp⟩ は，⟨exp⟩ の評価結果の値がポインタであれば，そのポインタが指す値を取り出す．⟨exp_1⟩ := ⟨exp_2⟩ は，⟨exp_1⟩ の評価結果の値がポインタであれば，そのポインタが指す値を，⟨exp_2⟩ の評価結果の値で書き換える．

⟨ty⟩ array の値は ⟨ty⟩ 型の値を要素の型とする変更可能な配列型である．

⟨ty⟩ array を含むその他の組み込み型を操作する組み込み関数は，型ごとに第 25 節で定義する標準ライブラリとして以下の名前のストラクチャにまとめられている．General ストラクチャには，参照型の組み込み演算，組み込み型を操作する際に発生するシステム定義の例外などが定義されている．

組み込み型	ストラクチャ	シグネチャ
`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

定義されている演算一覧は，以下のように対話型環境で対応するストラクチャを再定義してみると，見ることができる．

```
# 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 静的インポート式: _import ⟨*string*⟩ : ⟨*cfunty*⟩

C 関数を直接 SML#関数として使用するための式である．構文の構成要素の意味は以下のとおりである．

- ⟨*string*⟩：C 関数名．この名前が，リンク時にリンカによって外部名として参照される名前である．
- ⟨*cfunty*⟩：C 関数の型の指定．以下の文法に従い，C 関数の型を指定する．

C 関数型：⟨*cfunty*⟩

⟨*cfunty*⟩	::=	(⟨*cfunattr*⟩)? ⟨*argTyList*⟩ `->` ⟨*retTyOpt*⟩	
⟨*argTyList*⟩	::=	`(` ⟨*argTy*⟩ `,...,` ⟨*argTy*⟩ (`,` ⟨*varArgs*⟩)? `)`	(複数引数)
	|	⟨*argTy*⟩	(引数がただ 1 つ)
	|	`()`	(引数なし)
⟨*retTyOpt*⟩	::=	⟨*retTy*⟩	
	|	`()`	(C の `void` 型に対応)

コールバック関数型：⟨*argfunty*⟩

⟨*argfunty*⟩	::=	(⟨*cfunattr*⟩)? ⟨*retTylist*⟩ `->` ⟨*argTyOpt*⟩	
⟨*retTyList*⟩	::=	`(` ⟨*retTy*⟩ `,...,` ⟨*retTy*⟩ (`,` ⟨*varRets*⟩)? `)`	(複数引数)
	|	⟨*retTy*⟩	(引数がただ 1 つ)
	|	`()`	(引数なし)
⟨*argTyOpt*⟩	::=	⟨*argTy*⟩	
	|	`()`	(C の `void` 型に対応)

相互運用型：⟨*interoperableTy*⟩

⟨*interoperableTy*⟩	::=	(⟨*tySeq*⟩)? ⟨*longTycon*⟩	(C と受け渡し可能な型に限る．以下に詳述)

SML#から C に渡す引数の型：⟨*argTy*⟩

⟨*argTy*⟩	::=	⟨*argTy*⟩ `*` ⋯ `*` ⟨*argTy*⟩	(C の構造体へのポインタ型に相当)
	|	`{` ⟨*argTyRow*⟩ `}`	(C の構造体へのポインタ型に相当)
	|	⟨*tyvar*⟩	(`boxed` カインドを持つものに限る)
	|	⟨*interoperableTy*⟩	
	|	⟨*argfunty*⟩	(コールバック関数引数型)
⟨*argTyRow*⟩	::=	⟨*lab*⟩ `:` ⟨*argTy*⟩ (`,` ⟨*argTyRow*⟩)?	(⟨*lab*⟩ は ⟨*decimal*⟩ で始まるものに限る)

C から SML#に渡される返り値の型：⟨*retTy*⟩

⟨*retTy*⟩	::=	⟨*interoperableTy*⟩	
	|	⟨*tyvar*⟩	(`boxed` カインドを持つものに限る)

可変長引数型指定：⟨*varArgs*⟩ および ⟨*varRets*⟩

⟨*varArgs*⟩	::=	`...` `(` ⟨*argTy*⟩ `,...,` ⟨*argTy*⟩ `)`
⟨*varRets*⟩	::=	`...` `(` ⟨*retTy*⟩ `,...,` ⟨*retTy*⟩ `)`

C 関数属性指定：⟨*cfunattr*⟩

⟨*cfunattr*⟩	::=	`__attribute__((` ⟨*attr*⟩ `,...,` ⟨*attr*⟩ `))`
⟨*att*⟩	::=	`cdecl` | `stdcall` | `fastcc` | `pure` | `fast`

C 関数の型に現れる型名 ⟨*interoperableTy*⟩ は，以下の 2 つの条件を全て満たさなければならない．

1. ⟨*interoperableTy*⟩ は以下のいずれかでなければならない．
   - 相互運用可能な原子型：`int`, `int8`, `int16`, `int64`, `word`, `word8`, `word16`, `word64`, `real`, `real32`, `char`, または `string`.
   - C ポインタの型：`codeptr`, ⟨*interoperableTy*⟩ `ptr`, または `unit ptr`.
   - 多相 C ポインタ型：⟨*tyvar*⟩ `ptr` (ただし ⟨*tyvar*⟩ は `boxed` カインドまたは `unboxed` カインドを持つものに限る).

- サイズ型：⟨*ty*⟩ size.
- 配列型：⟨*interoperableTy*⟩ array, ⟨*interoperableTy*⟩ vector, または ⟨*interoperableTy*⟩ ref.
- 多相配列型：⟨*tyvar*⟩ array, ⟨*tyvar*⟩ vector, または ⟨*tyvar*⟩ ref (ただし ⟨*tyvar*⟩ は boxed カインドまたは unboxed カインドを持つものに限る).
- type 宣言で付けた，上記の型のいずれかの別名．(⟨*tySeq*⟩)? ⟨*longTycon*⟩ を展開すると上記のいずれかにならなければならない．ただし，⟨*argTy*⟩ としての ⟨*interoperableTy*⟩ に type 宣言で付けた型の別名が来た場合に限り，(⟨*tySeq*⟩)? ⟨*longTycon*⟩ を展開した結果が ⟨*argTy*⟩ 相当の組型あるいはレコード型であっても良い．

2. C から SML#に渡されたり，C が書き換えたりする可能性のある値の型に，string 型，array 型，vector 型，および ref 型が現れてはならない．すなわち，これらの型は以下の箇所に現れてはならない．

   - ⟨*retTy*⟩ としての ⟨*interoperableTy*⟩
   - array，ref，および ptr 型の型パラメタ

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 およびその仲間	同じサイズの符号付き整数型
word およびその仲間	同じサイズの符号付き整数型
real	double
real32	float
char	char
string	const char *
codeptr	C の関数へのポインタ型
τ ptr	τ へのポインタの型
unit ptr	void *または不完全型へのポインタ型
τ size	size_t
τ array	τ 型の配列の先頭を指すポインタ型
τ vector	τ 型の const 配列の先頭を指すポインタ型
τ ref	τ 型の要素数 1 の配列を指すポインタ型

SML#の int および word は常に 32 ビットサイズの整数であることに注意されたい．今日のほとんどのシステムでは SML#の int と C の int は対応するが，int が 32 ビットでないシステムではこれらは対応しない．

⟨*argTy*⟩ の ⟨$argTy_1$⟩ * ⋯ * ⟨$argTy_n$⟩ および{⟨lab_1⟩ : ⟨$argTy_1$⟩ , ⋯, ⟨lab_n⟩ : ⟨$argTy_n$⟩ }型は，⟨$argTy_1$⟩, ⋯, ⟨$argTy_n$⟩ をラベルの ⟨*decimal*⟩ の順番でメンバとして持つ const 付き構造体型へのポインタに対応する．もし，全ての ⟨$argTy_i$⟩ が同じならば，要素数 n の ⟨$argTy_i$⟩ の const 付き配列の先頭へのポインタにも対応する．

C 関数が SML#から見てパラメタ多相的に働くとき，かつその場合に限り，C 関数の引数および返り値の型として型変数を書いても良い．例えば，C の恒等関数

```
void *id(void *x) { return x; }
```

を以下のようにインポートしてもよい．

```
val 'a#boxed id = _import "id" : 'a -> 'a
```

また，任意のポインタの値を表示する printf 関数を以下のようにインポートしてもよい．

```
val 'a#boxed printPtr = _import "printf" : (string,...('a)) -> int
```

C 関数の型の前に記述する属性の意味は以下の通りである．

`cdecl` ターゲットプラットフォーム標準の C 関数呼び出し規約に従う C 関数であることを表す．呼び出し規約に関する属性が指定されていない場合のデフォルトである．

`stdcall` Windows プラットフォームにおける stdcall 呼び出し規約に従う C 関数であることを表す．

`fastcc` LLVM が提供する fastcc 呼び出し規約に従う C 関数であることを表す．

`pure` C 関数が SML#から見て**純粋な**関数であることを表す．すなわち，この属性を持つ C 関数は，メモリの書き換えを行わず，かつ引数リストのみから返り値がただ 1 つに定まる．SML#コンパイラの最適化に影響を与える．

`fast` C 関数が非常に短い時間で終了することを表す．この属性を持つ C 関数は，コールバック関数を通じて SML#のコードを呼び出したり，スレッドの実行を中断してはならない．SML#コンパイラは，この属性を持つ C 関数に対して，より効率の良い呼び出しコードを生成する．ただし，この属性を持つ C 関数が長い時間を消費すると，GC に影響を与え，この C 関数が終了するまでの間，全スレッドの実行が止まる可能性がある．

この式の型は，⟨*cfunty*⟩ で指定した C 関数の型に対応する SML#の関数型である．対応の定義は以下の通りである．

C 関数型	SML#関数型
(⟨*argTy*⟩ $_1$, ..., ⟨*argTy*⟩ $_n$ (⟨*varArgs*⟩)?) -> ⟨*retTy*⟩	⟨*argTy*⟩ $_1$ * ··· * ⟨*argTy*⟩ $_n$ (* ⟨*varArgs*⟩)? -> ⟨*retTy*⟩

⟨*argTyList*⟩ または ⟨*retTyOpt*⟩ が `()` の場合は，`unit` 型として SML#の型に現れる．⟨*interoperableTy*⟩，⟨*tyvar*⟩，および ⟨***⟩ はそのまま SML#の型に現れる．この定義は，コールバック関数型 ⟨*argfunty*⟩ についても同様である．

この式の値は，⟨*string*⟩ で指定された C 関数を呼び出す SML#の関数である．C 関数型の指定が正しい限り，通常の SML#関数として使用できる．

19.22 動的インポート式: ⟨*exp*⟩ : _import ⟨*cfunty*⟩

静的インポート式と同様の機能を，動的リンクによって得た関数ポインタに対して行う式である．⟨*exp*⟩ は `codeptr` 型でなければならない．この式を評価すると，⟨*exp*⟩ が評価され，関数ポインタを得る．関数ポインタが指すコードが ⟨*cfunty*⟩ で指示された型の C 関数ならば，この式の値は，⟨*exp*⟩ で指定された C 関数を呼び出す SML#の関数である．そうでない場合の式の意味は未定義である．

19.23 サイズ式 _sizeof(⟨*ty*⟩)

⟨*ty*⟩ 型の値のメモリ上での大きさ（バイト数）を表す定数式である．この式の型は ⟨*ty*⟩ `size` であり，値はバイト数を表す整数である．

サイズ式は主にインポートした C の多相関数を呼び出すのに用いる．例えば，C の標準ライブラリ関数 `memcpy` を配列の先頭 1 要素のコピーに用いる場合，インポート式は以下の通りである．

```
val 'a#unboxed memcpy =
    _import "memcpy" : ('a array, 'a vector, 'a size) -> unit ptr
```

このようにインポートした関数を以下のようにして呼び出すことができる．

```
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 動的型キャスト式 _dynamic ⟨exp⟩ as ⟨ty⟩

この式は動的型付けされた値 ⟨exp⟩ を ⟨ty⟩ にキャストする．⟨exp⟩ の型は τ Dynamic.dyn でなければならない．この式の型は ⟨ty⟩ である．

この式を評価すると，⟨exp⟩ が評価され，動的型付けされた値 v を得る．この式の値は，v の構造と ⟨ty⟩ に依存して決まる．その規則は以下の通りである．

- ⟨ty⟩ が Dynamic.void Dynamic.dyn のとき，この式の値は v である．
- ⟨ty⟩ が τ Dynamic.dyn 型のとき，v が τ 型のビューを持つ（v の部分構造を τ 型の値として取り出せる）ならば，この式の値は v である．そうでなければ Dynamic.RuntimeTypeError 例外を発生させる．
- ⟨ty⟩ が Dynamic.dyn 型を含まないとき，v の型が ⟨ty⟩ と一致するならば，この式の値は v を ⟨ty⟩ 型にキャストした値である．そうでなければ Dynamic.RuntimeTypeError 例外を発生させる．
- ⟨ty⟩ が Dynamic.dyn 型を含む型のとき，v および ⟨ty⟩ の構造に対して再帰的に以上の規則を適用する．

例えば，以下のようにして作ったレコードのリストに対して

```
val r = Dynamic.dynamic [{name = "Joe", age = 21}, {name = "Sue", age = 31}];
```

以下のキャストは全て正しい．

```
_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;
```

v 自体は必ずしも ML で型がつく構造でなくてもよい．例えば以下のようにして作ったヘテロジニアスなリストに対して

```
val l = Dynamic.fromJson
          "[{\"name\":\"Joe\", \"age\":21},\
          \{\"name\":\"Sue\", \"grade\":2.0},\
          \{\"name\":\"Robert\", \"nickname\":\"Bob\"}]";
```

以下のキャストは全て正しい．

```
_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;
```

この式を分割コンパイルモードで用いる場合，"reify.smi"を_require する必要がある．

19.25 動的型キャスト付き場合分け式 _dynamiccase ⟨*exp*⟩ of ⟨*match*⟩

⟨*exp*⟩ の評価結果の動的型付けされた値を動的型と値の構造に関する場合分けで処理する構文である．⟨*match*⟩ では，データ構成子を含むパターンとそのパターンにマッチした時に評価される式の組を以下の形で記述する．

```
  ⟨pat_1⟩ => ⟨exp_1⟩
| ...
| ⟨pat_n⟩ => ⟨exp_n⟩
```

各パターンの型はそれぞれ異なっていても良い．ただし，各パターンに現れる識別子パターンのうち変数のパターンおよび匿名パターンにはすべて型注釈が付けられていなければならない．また，これらパターン集合は以下の制約を満たさなければならない．

- 指定されたパターンをその型で分類したとき，各型のパターンの並びについて case 式と同様の制約を満たさなければならない．

例えば以下の例は int について冗長なためエラーとなる．

```
# fn x => _dynamiccase x of x:int => "int" | x:real => "real" | 0:int => "zero";
(interactive):1.8-1.76 Error: match redundant
      x => ...
  --> 0 => ...
```

この場合分け式の評価は以下のように行われる．⟨*exp*⟩ を評価し得られた動的型付けされた値 v を，⟨pat_1⟩ から ⟨pat_n⟩ のパターンに対して，この順にマッチングを試みる．マッチングの際，_dynamic 式と同様の方法で，v をパターンの型に動的型キャストする．キャストとマッチングの両方に成功した最初のパターン ⟨pat_i⟩ に含まれる変数を対応する値に束縛し，その束縛を現在の環境追加して得られる環境で，⟨pat_i⟩ に対応する式 ⟨exp_i⟩ を評価し，得られる値を，この式の値とする．動的型キャストに成功する型を持つパターンが存在しないときは Dynamic.RuntimeTypeError 例外が発生する．マッチするパターンが無い場合は Match 例外が発生する．

この式を分割コンパイルモードで用いる場合，"reify.smi"を_require する必要がある．

第20章　パターンとパターンマッチング

値束縛宣言（valBind）(23.1) の束縛対象および関数宣言（funDecl）(23.2)，case 式 (19.18)，fn 式 (19.19)，handle 式 (19.14) の引数には，パターンを記述する．変数はパターンの特殊な場合である．

パターンは，値が持つべき構造を記述する．記述されたパターンは，対応する値とマッチングが試みられ，マッチすれば，パターンに含まれる変数が，その現れる位置に対応する値に束縛される．パターンマッチングは，各パターンのスコープにある式を評価のための束縛環境にパターンに含まれる変数の束縛環境を追加する効果を持つ．従って，各パターンは，静的な評価の場合，追加すべき型環境を，実行時の動的な評価の場合，型環境に対応した値の環境を生成する．

$\langle pat \rangle$ 構文は以下の文法で与えられる．

- パターン（トップレベル）

$\langle pat \rangle$	::=	$\langle atpat \rangle$	
	|	(op)? $\langle longVid \rangle$ $\langle atpat \rangle$	データ構造
	|	$\langle pat \rangle$ $\langle vid \rangle$ $\langle pat \rangle$	データ構造（演算子表現）
	|	$\langle pat \rangle$: $\langle ty \rangle$	型制約パターン
	|	$\langle vid \rangle$ (: $\langle ty \rangle$)? as $\langle pat \rangle$	多層パターン

- 原子パターン

$\langle atpat \rangle$	::=	$\langle scon \rangle$	定数
	|	_	匿名パターン
	|	$\langle vid \rangle$	変数及びコンストラクタ
	|	$\langle longVid \rangle$	コンストラクタ
	|	{($\langle patrow \rangle$)? }	レコードパターン
	|	()	unit 型定数
	|	($\langle pat_1 \rangle$, $\cdots$, $\langle pat_n \rangle$)	組 ($n \geq 2$)
	|	[$\langle pat_1 \rangle$, $\cdots$, $\langle pat_n \rangle$]	リスト ($n \geq 0$)
	|	($\langle pat \rangle$)	

$\langle patrow \rangle$	::=	...	匿名フィールド
	|	$\langle lab \rangle$ = $\langle pat \rangle$ (, $\langle patrow \rangle$)?	レコードフィールド
	|	vid(: $\langle ty \rangle$)? (as $\langle pat \rangle$)? (, $\langle patrow \rangle$)?	変数兼ラベル

以下，各パターン $\langle pat \rangle$ について，その型 $\langle ty \rangle$ と生成される型環境 Γ およびマッチする値を定義する．実行時の動的な評価時に生成される環境は，生成される型環境に対応するマッチした値の束縛環境である．

定数パターン: $\langle scon \rangle$　19.2 で定義された定数式同一であり，対応する型を持ち，空の型環境 $\emptyset$ を生成する．同一の定数値にマッチする．

匿名パターン:_　文脈に従って決まる任意の型を持ち，空の型環境 $\emptyset$ を生成する．パターンの型を持つ任意の値にマッチする．

識別子パターン: ⟨*vid*⟩　この識別子名が，このパターンが現れる位置をスコープとして含むコンストラクタとして定義されていれば，そのコンストラクタの型を持ち，空の型環境を生成する．そのコンストラクタにマッチする．

識別子が定義されていないか，または，変数属性を持つ名前として定義されていれば，この名前は文脈によって定まる任意の型 ⟨*ty*⟩ を持ち，型環境 {⟨*vid*⟩ : ⟨*ty*⟩} を生成する．型 ⟨*ty*⟩ を持つ任意の値にマッチする．このパターンは，パターンを含む構文が定めるスコープ規則に従い，名前 ⟨*vid*⟩ が再定義される．

以下に簡単な例を示す．

```
SML# 3.7.1 (2021-03-15) for x86_64-pc-linux-gnu with LLVM 11.0.0
# 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 識別子パターン: ⟨*longVid*⟩　この long 識別子名が，このパターンが現れる位置をスコープとして含む引数を持たないコンストラクタ ⟨*vid*⟩ として定義されていれば，空の型環境が生成されれ，コンストラクタ ⟨*vid*⟩ にマッチする．現在の環境に long 識別子がコンストラクタとして定義されていないか，または，引数を持つコンストラクタとして定義されていれば，エラーとなる．以下に簡単な例を示す．

```
# 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` は引数を持つコンストラクタであり，`A.C` は変数であるため，関数宣言 `g` と `h` はエラーとなる．

レコードパターン: ⟨{(⟨*patrow*⟩)? }⟩　レコードフィールドパターン ⟨*patrow*⟩ が指定されない`{}`の形のパターンの場合，空のレコード`{}`にマッチする．変数の型環境は生成されない．

レコードフィールドパターン ⟨*patrow*⟩ が指定されている場合，そのフィールド型を ⟨lab_1⟩ : ⟨ty_1⟩ ,..., ⟨lab_n⟩ : ⟨ty_n⟩，生成される変数の型環境を Γ とする．レコードフィールドパターン ⟨*patrow*⟩ の形に応じて以下の 2 通りの場合がある．

1. 匿名フィールド... を含まない固定レコードパターンの場合．フィールド型からなる単相レコード型 {⟨lab_1⟩ : ⟨ty_1⟩ ,..., ⟨lab_n⟩ : ⟨ty_n⟩} を持ち，型環境 Γ を生成する．

   このパターンは，レコードフィールドパターン ⟨$patrow$⟩ にマッチするレコードフィールドを含む上記の型のレコードにマッチする．

2. 多相レコードパターン．レコードフィールドパターン ⟨$patrow$⟩ が，匿名フィールド... を含む場合．⟨$patrow$⟩ のフィルド型をレコードカインドとしてもつ多相レコード型'a#{⟨lab_1⟩ : ⟨ty_1⟩ ,..., ⟨lab_n⟩ : ⟨ty_n⟩} を持ち，レコードフィールドパターン ⟨$patrow$⟩ が生成する変数束縛を生成する．この多相型は，文脈の型制約により，種々の多相レコード型または単相レコード型にインスタンス化されうる．

   このパターンは，レコードフィールドパターン ⟨$patrow$⟩ にマッチするレコードフィールドを含む上記の型のレコードにマッチする．

レコードフィールドパターン: ⟨$patrow$⟩

- 匿名フィールドパターン：...．指定されたフィールド集合が，レコードの一部であることの指定である．
- フィールドパターン: ⟨lab⟩ = ⟨pat⟩ (, ⟨$patrow$⟩)?. (, ⟨$patrow$⟩)? のフィールド型を ⟨lab_1⟩ : ⟨ty_1⟩ ,..., ⟨lab_n⟩ : ⟨ty_n⟩, 型環境を Γ，パターン ⟨pat⟩ の型を ⟨ty⟩，生成する型環境を Γ_0 とする．

  ラベル ⟨lab⟩ が ⟨lab_1⟩ ,..., ⟨lab_n⟩ とすべて異なる場合，フィールド型 ⟨lab⟩ : ⟨ty⟩ , ⟨lab_1⟩ : ⟨ty_1⟩ ,..., ⟨lab_n⟩ : ⟨ty_n⟩, を持ち，型環境 $\Gamma_0 \cup \Gamma$ が生成される．ラベル ⟨lab⟩ が ⟨lab_1⟩ ,..., ⟨lab_n⟩ の何れかと一致する場合は型エラーである．
- 変数兼ラベルパターン：vid(: ⟨ty⟩)? (as ⟨pat⟩)? (, ⟨$patrow$⟩)?.

  以下のフィールドパターンに変換された後評価される．

$$\text{vid} = \text{vid}(: \langle ty \rangle)?\ (\texttt{as}\ \langle pat \rangle)?\ (,\ \langle patrow \rangle)?.$$

  生成される型環境，マッチする値は，対応するレコードパターンの場合と同一である．以下に簡単な例を示す．

```
# 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
```

組パターン: (⟨pat_1⟩ ,···, ⟨pat_n⟩)　レコードパターン

$$\texttt{\{1 =}\ \langle pat_1 \rangle\ \texttt{,}\ \cdots\texttt{,}\ n\ \texttt{=}\ \langle pat_n \rangle\ \texttt{\}}$$

に変換され，評価される．生成される型環境，マッチする値は，対応するレコードパターンの場合と同一である．以下に簡単な例を示す．

```
# 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
```

リストパターン: `[` $\langle pat_1 \rangle$ `,` $\cdots$ `,` $\langle pat_n \rangle$ `]`　ネストしたリストデータ構造パターン

$$\langle pat_1 \rangle \texttt{ :: } \cdots \texttt{ :: } \langle pat_n \rangle \texttt{ :: nil}$$

に変換され，評価される．生成される型環境，マッチする値は，対応するコンストラクタアプリケーションパターンの場合と同一である．以下に簡単な例を示す．

```
# 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
```

データ構造パターン: (op)? $\langle longVid \rangle$ $\langle atpat \rangle$　$\langle longVid \rangle$ が型 $\langle ty_1 \rangle$ `->` $\langle ty_2 \rangle$ を持つコンストラクタ C に束縛されている時，型 $\langle ty_2 \rangle$ を持ち，$\langle atpat \rangle$ が生成する型環境を生成する．このパターンは，$\langle atpat \rangle$ のマッチするデータ v を部分構造としてもつデータ構造 $C(v)$ にマッチする．

演算子表現のデータ構造パターン $\langle pat_1 \rangle$ $\langle vid \rangle$ $\langle pat_2 \rangle$ は，`op` $\langle vid \rangle$ `(` $\langle pat_1 \rangle$ `,` $\langle pat_2 \rangle$ `)` に変換された後評価される．その型と生成される型環境，マッチする動的値は，変換後のパターンと同一である．

型制約パターン: $\langle pat \rangle : \langle ty \rangle$　パターン $\langle pat \rangle$ が型 $\langle ty_1 \rangle$ と型環境 Γ_0 を持ちかつ $\langle ty_1 \rangle$ と $\langle ty \rangle$ が型代入 S のもとで単一化できる場合，型 $S($ $\langle ty \rangle$ $)$ を持ち，型環境 $S(\Gamma_0)$ をもつ．型制約の下で，パターン $\langle pat \rangle$ にマッチする値にマッチする．

多層パターン: $\langle vid \rangle$ (: $\langle ty \rangle$)? `as` $\langle pat \rangle$　パターン $\langle pat \rangle : \langle ty \rangle$ が型 $\langle ty' \rangle$ と型環境 Γ をもてば，このパターンは型 $\langle ty' \rangle$ と型環境 $\Gamma \cup \{x : \langle ty' \rangle\}$ をもつ．このパターンは，$\langle pat \rangle : \langle ty \rangle$ にマッチする動的値にマッチする．

第21章　識別子のスコープ規則

本章では，プログラムに使用される名前の有効範囲（スコープ規則）を定義する．
17.2 節で定義した通り，識別子は，以下の 7 種類のクラスの名前として使用される．

識別子名	名前クラス
⟨*vid*⟩	変数とデータコンストラクタ名
⟨*strid*⟩	ストラクチャ名
⟨*sigid*⟩	シグネチャ名
⟨*funid*⟩	ファンクタ名
⟨*tycon*⟩	型構成子名
⟨*tyvar*⟩	型変数名
⟨*lab*⟩	レコードラベル

この中で，'... の形をした型変数名は，他の名前とは字句構造上区別される．それ以外の名前の字句構造は重なりを持つ．例えば A は，型変数名以外の他の全ての名前として使用される．第 21 章で定義する通り，これら同一の識別子のクラスは，それがプログラム中に現れる位置によって一意に定まるように，文法が定義されている．さらにこれらの名前は，そのクラス毎に独立な名前空間で管理されるため，各クラスの名前として同一の字句を使用することができる．以下は，名前の使用の例である．

```
(* 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
```

7 行目の最初の A は 1 行目の変数 A を，２番目の A は２行目の型構成子 A を参照しており，6 行目の最初の A は５行目のストラクチャを参照している．

レコードラベル以外の名前は，プログラムの構文によって定義され，その定義が有効な範囲で参照される．名前の定義を含む構文とそれによって定義される名前は以下の通りである．

1. **分割コンパイルのインタフェイスファイル**．インタフェイスファイルに含まれるプロバイドリストは名前を定義する．例えば，インタフェイスファイルに

   ```
   _require "myLibrary.smi"
   val x : int
   datatype foo = A of int | B of bool
   ```

   の記述があれば，変数 x，データ構成子 A,B，型名 foo が定義される．

2. **ストラクチャ宣言**．ストラクチャ名とストラクチャ内の各宣言が定義する名前にストラクチャ名をプリフィックスしたロング名が定義される．例えば，ストラクチャ式 S がロング識別子名集合 L を定義する場合，ストラクチャ宣言 `structure S =` S によってストラクチャ名 S および，ロング名の集合 $\{S.path \mid path \in L\}$ が定義される．

3. **ファンクタ宣言**．ファンクタ名とファンクタの引数の宣言仕様の各名前が定義される．例えば`functor F(type foo) = ··`によってファンクタ名`F`と型構成子名`foo`が定義される．
4. **型構成子の別名宣言**．型構成子名と型構成子の型変数引数が定義される．例えば`type foo = ···`によって型名（引数を持たない型構成子）`foo`が定義される．
5. **データ型宣言**．型構成子名と型構成子の型変数引数，データ構成子名が定義される．例えば`datatype 'a foo = A of in`によって型名`foo`，型変数`'a`，データ構成子名`A`，`B`が定義される．
6. **例外宣言**．例外構成子名が定義される．例えば`exception E`によって例外名`E`が定義される．
7. **値束縛宣言**．束縛パターンに含まれる変数が定義される．例えば，`x`がデータ構成子として定義されていない位置で`val x = 1`と宣言すると変数`x`が定義される．
8. **関数宣言**．関数名と関数の引数パターンに含まれる変数が定義される．例えば，`x`がデータ構成子として定義されていない位置で`fun f x = 1`と宣言すると変数`f`が定義される．
9. `fn` **式**．引数パターンに含まれる変数が定義される．例えば，`x`がデータ構成子として定義されていない位置で`fn x => x`では，変数`x`が定義され使用されている．
10. `case` **式**．ケースパターンに含まれる変数が定義される．例えば，`x`がデータ構成子として定義されていない位置で`case y of A x => x`では変数`x`が定義され使用されている．
11. **SQL 式**．`_sql`から始まる形の SQL 式には，変数定義が含まれるものがある．これらは第 22 章で定義する．

これら定義された名前の定義は，それぞれの名前空間毎に，その名前定義が現れる構文によって決まるスコープ（変数が参照できる範囲）を持つ．ALGOL 系ブロック構造言語の伝統を受け継ぐ SML#言語では，スコープを区切る構文は入れ子構造を成すため，名前定義のスコープも一般に入れ子状に重なりをもつ．各名前定義のスコープは，内側に現れる同一種類の同名の変数のスコープを除いた部分と定義される．

SML#言語のスコープを区切る構文とそれら構文に現れる名前の定義のスコープ規則を定義する．実際のスコープは，以下定義されるスコープから，内側に現れる同種類の同名の名前のスコープを除いた部分である．

- 分割コンパイル単位．

  分割コンパイル単位は一つのソースファイル ⟨*srcFule*⟩`.sml` である．

  ⟨*srcFile*⟩`.sml` のトップレベルの宣言に含まれる名前定義のスコープは，それに続くファイル全体である．

  さらに，このソースファイルに対応するインターフェイスファイル ⟨*srcFile*⟩`.smi` が定義する名前のスコープは，ソースファイル全体である．インターフェイスファイル ⟨*srcFile*⟩`.smi` で定義される名前は，インターフェイスファイルに直接含まれる各`_require`文で参照されるインターフェイスファイルが定義するすべての名前である．

- ストラクチャ生成文．

  `struct ··· end` のトップレベルの宣言に含まれる名前定義のスコープは，その定義に続く `end` までの部分である．

- `local` 構文．

  構文 `local` ⟨$decl_1$⟩ `in` ⟨$decl_2$⟩ `end` の ⟨$decl_1$⟩ の中の宣言に含まれる名前のスコープは，その宣言以降の ⟨$decl_1$⟩ の部分および ⟨$decl_2$⟩ 全体である．⟨$decl_2$⟩ の中の宣言に含まれる名前のスコープは，その宣言以降の ⟨$decl_2$⟩ の部分である．

- `let` 構文.

  構文 `let` ⟨*decl*⟩ `in` ⟨*exp*⟩ `end` の ⟨*decl*⟩ の中に現れる宣言に含まれる名前定義のスコープはその宣言以降の ⟨*decl1*⟩ の部分および ⟨*exp*⟩ である.

- `fun` 宣言

  構文

  $$\begin{array}{l}\texttt{fun}\ \langle id\rangle\ \ \langle pat_{1,1}\rangle\ \cdots\ \langle pat_{1,n}\rangle\ \texttt{=}\ \langle exp_1\rangle \\ \qquad\vdots \\ \ \ \texttt{|}\ \langle id\rangle\ \ \langle pat_{m,1}\rangle\ \cdots\ \langle pat_{m,n}\rangle\ \texttt{=}\ \langle exp_m\rangle\end{array}$$

  において，関数名 ⟨*id*⟩ のスコープは，⟨exp_1⟩ から ⟨exp_m⟩ の各式，パターン ⟨$pat_{i,j}$⟩ に含まれる変数定義のスコープは，対応する ⟨exp_i⟩ である.

- `fn` 式

  構文

  $$\texttt{fn}\ \langle pat_1\rangle\ \texttt{=>}\ \langle exp_1\rangle\ \ \texttt{|}\ \ \cdots\ \ \texttt{|}\ \langle pat_n\rangle\ \texttt{=>}\ \langle exp_m\rangle$$

  において，パターン ⟨pat_i⟩ に含まれる変数定義のスコープは，対応する ⟨exp_i⟩ である.

- `case` 式

  構文

  $$\texttt{case}\ exp\ \texttt{of}\ \langle pat_1\rangle\ \texttt{=>}\ \langle exp_1\rangle\ \ \texttt{|}\ \ \cdots\ \ \texttt{|}\ \langle pat_n\rangle\ \texttt{=>}\ \langle exp_m\rangle$$

  において，パターン ⟨pat_i⟩ に含まれる変数定義のスコープは，対応する ⟨exp_i⟩ である.

第22章　SQL式とコマンド

SML#では，標準SQLに含まれる代表的なクエリ構文の多くをそのままSML#の式としてプログラム中に書くことができる．SQLクエリそのものや，それを構成する部分式は，SML#では第一級のオブジェクトである．すなわち，それらにはSML#の型が与えられ，ML式と同様に，SML#の型検査の枠組みの下で整合性が検査される．また，型が合っている限り，SQLとSML#の他の言語機能とを自由に組み合わせることができる．例えば，部分的に構築されたSQLクエリやSQL式をデータとして取っておき，それらを動的に組み合わせて完全なSQLクエリを作ることができる．

22.1　SQLの型

22.1.1　SQLの基本型

NULLを除いて，以下のSML#の基本型がSQLの基本型にそれぞれ対応する．

SML#の基本型	対応するSQLの型
`int`, `intInf`, `word`	整数型
`bool`	SQL:99（機能ID T031）のBOOLEAN型
`char`	`CHAR(1)`型
`string`	`TEXT`または`VARCHAR`型
`real`	倍精度浮動小数点型
`real32`	単精度浮動小数点型

さらに，以下の数値型がSQLとの相互運用のために定義されている．

SML#の型	対応するSQLの型
`SQL.numeric`	`NUMERIC`型（最大精度の10進数）
`SQL.decimal`	`DECIMAL`型（`NUMERIC`型の別名）

SML#の型とデータベース側の型の具体的な対応付けは，接続先のデータベースエンジンと，SML#におけるそのデータベースエンジンのサポート状況に依存する．データベースエンジンごとの具体的な型の対応は22.8.1節で示す．

SQLのNULLは，SML#では`option`型の`NONE`に対応付けられる．NOT NULL制約が無いカラムの参照など，NULLを返す可能性がある式の型は，上記の型のいずれかの`option`型である．

22.1.2　SQLの論理演算式の型

`SQL.bool3`型は，比較演算子や論理演算子からなるSQLの論理演算式の型である．この`SQL.bool3`型は，SQLクエリの静的な整合性検査のために，真偽値に評価されるSQL評価式の断片に便宜上付けた型であり，どの標準SQLの型とも対応しない．

論理演算式の型`SQL.bool3`と真偽値の型`bool`を区別していることは，SQLの真偽値の取り扱いに関して歴史的な混乱があることに由来している．SQLでは伝統的に，真偽判定は真，偽，不明の3値で行われ，真偽値は第一級ではなく（例えば真偽値をテーブルに保存することはできない），真偽値のリテラルも存在しなかった．第一級の真偽値型BOOLEANは，SQL99において，オプショナルな機能（機能ID T031）として導入された．しかし，このオプショナルな規格にはSQLの他の部分との間で技術的な不整合があることが指摘されており，結局，このBOOLEAN型はRDBMSベンダーからの支持をほとんど集めなかっ

た．規格策定から 20 年近く立った現在においても，主要な RDBMS は，唯一の例外である PostgreSQL を除いて，BOOLEAN 型を提供していない．

この混乱から来る誤解や非互換性を避けるため，SML#は論理演算式と真偽値の区別を型上で強制する．真理値リテラル `true` および `false` は論理演算式として使用することはできず，また論理演算の結果を BOOLEAN 値として取り出すことはできない．真偽値リテラルについては 22.4.2 節を，SQL 論理演算式については 22.4.6 節を参照せよ．

22.1.3　SQL のテーブルおよびスキーマの型

SQL のテーブル，ビュー，およびスキーマには，SML#のレコードのリスト型が対応付けられる．SQL のテーブルおよびビューには，カラムの名前と型をそれぞれフィールドラベルおよびそのフィールドの型としたレコード型のリスト型が与えられる．カラムに NOT NULL 制約または PRIMARY KEY 制約が付いていないとき，そのカラムの型はいずれかの基本型の `option` 型である．例えば，

```
CREATE TABLE foo (bar INT, baz TEXT NOT NULL);
```

と定義されたテーブル `foo` の構造は，SML#では

```
{bar : int option, baz : string} list
```

という型で表現される．

このように，テーブルあるいはカラムに付く制約のうち，NOT NULL 制約および PRIMARY KEY 制約の「NULL ではない」ことだけは SML#の型で表現され，SML#の型システムによって静的に検査される．その他の制約は SQL データベースによってクエリ実行時に検査される．

SQL スキーマは，テーブルおよびビューの名前をフィールドラベル，その構造を表す型をフィールド型とするレコード型で表現される．例えば，

```
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);
```

と定義されたスキーマは，SML#では

```
{
  employee : {id : int, name : string, age : int,
              deptId : int option, salary : int option} list,
  department : {deptId : int, name : string} list
}
```

と表現される．

22.1.4　SQL クエリおよびその断片の型

SQL クエリおよびその部分式は，その構文カテゴリに対応して，それぞれ異なる型を持つ．SQL の各構文カテゴリとその型は以下の通りである．

構文カテゴリ	その項の型
SQL 評価式	(τ_1 -> τ_2,w) SQL.exp
SQL コマンド	(τ,w) SQL.command
SELECT クエリ	(τ,w) SQL.query
SELECT 句	(τ_1,τ_2,w) SQL.select
FROM 句	(τ,w) SQL.from
WHERE 句	(τ,w) SQL.whr
ORDER BY 句	(τ,w) SQL.orderby
OFFSET 句	(τ,w) SQL.offset
LIMIT 句	(τ,w) SQL.limit

ここで，τ はテーブルの型あるいは基本型，w は接続先データベースを識別する型である．各型の意図は以下の通りである．

- (τ_1 -> τ_2，w) SQL.exp は，データベース接続 w および τ_1 型の行の下で τ_2 型を持つ SQL 評価式の型である．
- (τ_1,τ_2,w) SQL.select は，データベース接続 w の下で τ_1 型のテーブルを τ_2 型のテーブルに変換する SELECT 句の型である．
- これら以外の構文カテゴリの型 (τ,w) SQL.X は，データベース接続 w の下で τ 型を持つ X の型である．

22.1.5 SQL 関連のハンドルの型

SQL データベースへの接続や，SQL クエリの結果に対して，SML#は以下の型を与える．

型	説明
τ SQL.server	τ 型のスキーマを持つデータベースを管理するサーバーへの接続先
τ SQL.conn	τ 型のスキーマを持つデータベースを管理するサーバーへの接続ハンドル
τ SQL.cursor	τ 型のテーブルにアクセスするためのカーソル
(τ,w) SQL.db	接続 w で接続されている τ 型のスキーマを持つデータベースの実体

SQL を用いる典型的な SML#プログラムでは，これらの型はおおよそ以下のように使用される．

1. _sqlserver 構文を用いて，τ SQL.server 型の接続先情報を作る．(22.3 節参照)
2. SQL.connect 関数を用いて，τ SQL.server 型の接続先に接続し，τ SQL.conn 型の接続ハンドルを得る．(22.8.1 節参照)
3. SQL クエリを，['a. (τ,'a) SQL.db -> (τ' SQL.cursor,'a) SQL.command] 型を持つ多相関数として構築する（22.1.6 節参照）．
4. _sql 構文を用いて，SQL クエリを τ SQL.conn -> τ' SQL.cursor 型の関数に変換する（22.7 節参照）．
5. この関数を τ SQL.conn 型の接続ハンドルを引数として呼び出すと，SQL クエリがサーバーに送信され，クエリがサーバーで評価される．クエリの評価に成功すると，評価結果にアクセスする τ' SQL.cursor 型のカーソルが得られる（22.7 節参照）．
6. SQL.fetch 関数または SQL.fetchAll 関数を用いて τ' SQL.cursor 型のカーソルから τ' 型のレコードを取得する（22.8.2 節参照）．

22.1.6 SQL 式の型付け方針

SML#の SQL 式は，これまでに定義した基本型の対応，論理演算式型の導入，テーブル構造とレコードのリスト型の対応付けを用いて型付けされる．SQL クエリは，それが行うテーブル処理と同じことをレコードのリストに対して行う SML#のプログラムと同型の型を持つ．

例えば，

```
SELECT t.name AS employeeName, t.age AS employeeAge
FROM employeeTable AS t
WHERE t.age > 20
```

という SQL クエリを，SML#では

```
val Q = fn db => _sql select #t.name as employeeName, #t.age as employeeAge
                      from #db.employeeTable as t
                      where #t.age > 20
```

と書く．元の SQL には現れない変数 db は，クエリが実行される対象のデータベースを抽象する変数である．

さて，このクエリは，FROM 句が参照するテーブルの各行を WHERE 句の条件でフィルタにかけ，残った各行を SELECT 句の式で変形したテーブルを計算するクエリである．このクエリと同じことを，SQL のテーブルの代わりに SML#のレコードのリストに対して行う SML#プログラムを単純に書くと，以下のようになる．

```
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)))
```

この関数 Q' の型は

```
val Q' : ['a#{employeeTable : 'b list},
          'b#{age : int, name : 'c},
          'c.
          'a -> {employeeAge : 'c, employeeName : int} list]
```

である．この関数 Q' のことを，SQL クエリ Q のトイプログラムと呼ぶ．

この SQL クエリと SML#式の対応を通じて，上述のクエリ Q には,「このリストを扱う SML#プログラムと同等のことをデータベースサーバーが実行する SQL クエリ」を表す以下の型が与えられる．

```
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]
```

この Q の型に現れるカインド付き型変数は，このクエリの以下の性質をそれぞれ表している．

- 'a は，このクエリが対象とするデータベースには少なくとも'b 型の employeeTable テーブルがなくてはならないことを表す．それ以外のテーブルの存在はこのクエリの評価に関係しない．

- 'b は，employeeTable テーブルには少なくとも int 型の age カラムと'c の型の name カラムがなくてはならないことを表す．それ以外のカラムが employeeTable テーブルにあってもよい．
- 'c は，name カラムの型は SQL 基本型の範囲で任意であることを表す（実際には，'c のカインドから参照される型変数がもうひとつ存在するが，ここでは省略する）．
- 'd は，このクエリが任意の接続ハンドルを通じてサーバーに送信できることを表す（1 つのクエリが複数の送信先にまたがって書かれていないことを検査するために用いられる）．

Q にレコード多相型が付いていることから分かる通り，SQL クエリはデータベースに対して多相的である．SML#は，対象のデータベースが抽象された SQL クエリに対して，その SQL クエリが持つ最も一般的な型を推論する．

SQL クエリ全体と同様に，SQL クエリのいくつかの断片についても，レコードとリストを扱う SML#式との自然な対応を考えることができる．本章で定義する SQL 関連構文の型付け規則はすべて，自然な対応に従って定められている．

22.2 SQL クエリのための ML 式の拡張構文

第 19 章で定義されている通り，SML#では SQL クエリのために以下の拡張構文を導入している．

⟨*exp*⟩	::=	···	
	|	`_sqlserver` (⟨*appexp*⟩)? : ⟨*ty*⟩	SQL サーバ
	|	`_sql` ⟨*pat*⟩ `=>` ⟨*sqlfn*⟩	SQL 実行関数
	|	`_sql` ⟨*sql*⟩	SQL クエリ断片
⟨*atexp*⟩	::=	···	
	|	`_sql` (⟨*sql*⟩)	SQL クエリ断片

⟨*sql*⟩ および ⟨*sqlfn*⟩ は以下の通りである．

⟨*sql*⟩	::=	⟨*sqlexp*⟩	SQL 評価式
	|	⟨*sqlselect*⟩	SELECT クエリ
	|	⟨*sqlclause*⟩	SQL クエリ句
	|	⟨*sqlcommand*⟩	SQL コマンド
⟨*sqlfn*⟩	::=	⟨*sqlselect*⟩	SELECT クエリ
	|	⟨*sqlclause*⟩	SQL クエリ句
	|	⟨*sqlcommand*⟩	SQL コマンド

⟨*sqlexp*⟩ は 22.4 節で，⟨*sqlselect*⟩ および ⟨*sqlclause*⟩ は 22.5 節で，⟨*sqlcommand*⟩ は 22.6 節で，それぞれ定義する．

これらの構文のうち `_sql` で始まる式に現れる ⟨*pat*⟩ および ⟨*sql*⟩ の中では，...(⟨*exp*⟩) または (... ⟨*exp*⟩) という形で囲われる ⟨*exp*⟩ の中を除き（22.4 節，22.5 節，および 22.6 節参照），以下は予約語として扱われる（この予約語のリストは第 17.2 節からの再掲である）．これらの予約語を SQL 予約語と呼ぶ．

```
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
```

また，これらの構文には以下の制限がある．

- `_sql` に続く ⟨*pat*⟩ は開き丸括弧 "(" から始まってはならない．
- 式（トップレベル）の `_sql` ⟨*sql*⟩ の ⟨*sql*⟩ は，SQL 予約語で始まらなければならない．
- 式（トップレベル）の `_sql` ⟨*sql*⟩ は，以下の位置以外に現れてはならない．

- val や fun 宣言の=の右側
- let 式の in と end の間
- 逐次実行式 (···;···) の中
- 組ではない式を囲う丸括弧の中
- 上記 4 つの位置にある fn 式の=>の右側

22.3 接続先データベース宣言式：_sqlserver

_sqlserver 式は，接続先データベースサーバーの情報とそのサーバーが管理するデータベースのスキーマを表す．_sqlserver 式の評価結果を引数として SQL.connect 関数（22.8.1 節参照）を呼び出すと，_sqlserver 式が表すサーバーへの接続が試みられ，接続が確立し，かつ接続先データベースが_sqlserver 式が表すスキーマを包含するデータベースを管理していることが確認できたならば，そのサーバーへの接続ハンドルが得られる．

_sqlserver 式には，接続先データベースの種類，データベース固有の接続パラメタ，およびデータベーススキーマを表すレコード型をこの順に書く．_sqlserver 式の典型的な書き方の例を以下に示す．

```
_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
    }
```

この例は，localhost:5432 で待ち受けている PostgreSQL サーバーが，このレコード型と同型のスキーマを持つデータベースを管理していることを表している．サポートするデータベースの種類や接続パラメタの詳細については 22.8.1 節を参照せよ．

なお，実際には，_sqlserver と:の間には任意の関数適用式 ⟨*appexp*⟩ を書くことができる．この例においても，SQL.postgresql は実際は関数であり，"host=localhost port=5432"はその関数の引数である．この位置に任意の SML#の式を書くことで，同じスキーマを持つ異なる接続先のひとつを実行時に選ぶようなプログラムを書くことも可能である．式 ⟨*appexp*⟩ の型は SQL.backend でなければならない．

_sqlserver ⟨*appexp*⟩ : ⟨*ty*⟩ の型は，⟨*ty*⟩ SQL.server である．⟨*ty*⟩ は，22.1.3 節で示した，データベーススキーマを表すレコード型でなければならない．

過去のバージョンとの互換性のため，以下の書き方が許されている．これらの記法を新規のプログラムで使うことは推奨されない．

- ⟨*appexp*⟩ を省略することができる．省略した場合は，PostgreSQL サーバーにデフォルトの接続パラメタで接続することを表す．
- ⟨*appexp*⟩ に文字列リテラルだけを書ける．その文字列は PostgreSQL サーバーへの接続パラメタと解釈される．

22.4 SQL 評価式

SQL 評価式（value expression）は，SQL コマンドの中で値に評価される式である．SML#の SQL 評価式は，サーバーで評価される式を構築する式である．SML#の SQL 評価式として書いたほぼそのままの内容が，サーバーに送信される SQL 評価式となる．SQL 評価式自体の評価はデータベースサーバーで行われる．例えば，

```
_sql(1 + #employee.salary)
```

という SML#の SQL 評価式は，サーバーに送信する SQL クエリの断片

```
1 + employee.salary
```

に評価される．

SQL 評価式が SML#で評価できる部分式を含んでいる場合，その部分式は SQL 評価式構築時に評価され，評価された値がサーバーに送信する SQL クエリに埋め込まれる．例えば，

```
_sql(1 + 2 + #employee.salary)
```

という SML#の SQL 評価式は，サーバーに送信する SQL クエリの断片

```
3 + employee.salary
```

に評価される．

SML#では以下に階層的に定義される標準 SQL のサブセットを含む式 ⟨*sqlexp*⟩ を SQL 評価式として使用することができる．

- SQL 評価式（トップレベル）

  ⟨*sqlexp*⟩ ::= ⟨*sqlinfexp*⟩
  | `not` ⟨*sqlexp*⟩ SQL の論理否定
  | ⟨*sqlexp*⟩ `and` ⟨*sqlexp*⟩ SQL の論理積
  | ⟨*sqlexp*⟩ `or` ⟨*sqlexp*⟩ SQL の論理和

- SQL 演算子式

  ⟨*sqlinfexp*⟩ ::= ⟨*sqlcastexp*⟩
  | ⟨*sqlinfexp*⟩ ⟨*vid*⟩ ⟨*sqlinfexp*⟩ 二項演算

- SQL 型キャスト式

  ⟨*sqlcastexp*⟩ ::= ⟨*sqlappexp*⟩
  | `(` ⟨*vid*⟩ `)` ⟨*sqlcastexp*⟩ 型キャスト

- SQL 関数適用式

  ⟨*sqlappexp*⟩ ::= ⟨*sqlatexp*⟩
  | ⟨*vid*⟩ ⟨*sqlatexp*⟩ 関数適用
  | ⟨*sqlappexp*⟩ ⟨*sqlatexp*⟩ SML#の関数適用
  | ⟨*sqlappexp*⟩ `is` (`not`)? ⟨*sqlis*⟩ SQL の IS 述語

  ⟨*sqlis*⟩ ::= `null` | `true` | `false` | `unknown`

- SQL 原始式

⟨*sqlatexp*⟩	::=	⟨*scon*⟩	SML#の定数
	|	`true`	SML#の true リテラル
	|	`false`	SML#の false リテラル
	|	`null`	SQL の NULL リテラル
	|	`#` ⟨*lab*⟩ `.` ⟨*lab*⟩	名前のある関係に属するカラムの参照
	|	`#.` ⟨*lab*⟩	名前のない関係に属するカラムの参照
	|	⟨*vid*⟩	SML#の変数参照
	|	`op` ⟨*longvid*⟩	SML#の変数参照
	|	`(` ⟨*sqlexp*⟩ `,` ⋯ `,` ⟨*sqlexp*⟩ `)`	組
	|	`((_sql)?` ⟨*sqlselect*⟩ `)`	SQL の SELECT サブクエリ
	|	`(_sql)? exists ((_sql)?` ⟨*sqlselect*⟩ `)`	SQL の EXISTS サブクエリ
	|	`((_sql)?` ⟨*sqlcommand*⟩ `)`	
	|	`((_sql)?` ⟨*sqlclause*⟩ `)`	
	|	`(` ⟨*sqlexp*⟩ `)`	
	|	`(` ... ⟨*exp*⟩ `)`	SQL 評価式の埋め込み

他の SQL 関連構文との見た目上の統一感を出すために，一部のネストした SQL 評価式は予約語`_sql` から始めても良い．これらの`_sql` は単純に無視される．

SQL 評価式 ⟨*sqlexp*⟩ の型 τ は，静的な文脈として与えられるテーブル集合の型 τ' およびデータベース接続を識別するための型 w の下で決定される．ひとつの SQL 評価式に含まれる全ての部分式の型は，同じ τ' および w の下で与えられる．以下，式 e が τ' および w の下で型 τ を持つことを，e の型は $(\tau', w) \triangleright \tau$ 型である，と言う．τ' および w に言及する必要がない場合は単に，e の型は τ である，と言う．

22.4.1 SML#で評価される式

以下の再帰的な条件を満たす SQL 評価式は SML#で評価され，その値がデータベースサーバーに送信されるクエリに埋め込まれる．

1. 定数式 ⟨*scon*⟩ は SML#で評価される．
2. 変数式 ⟨*vid*⟩ および `op` ⟨*longvid*⟩ は SML#で評価される．
3. 組 `(` ⟨$sqlexp_1$⟩ `,` ⋯ `,` ⟨$sqlexp_n$⟩ `)` は，全ての ⟨$sqlexp_i$⟩ が SML#で評価される式ならば，SML#で評価される．
4. 関数適用式 ⟨*vid*⟩ ⟨*sqlatexp*⟩ は，⟨*sqlatexp*⟩ が SML#で評価される式ならば，SML#で評価される．
5. 二項演算式 ⟨$sqlinfexp_1$⟩ ⟨*vid*⟩ ⟨$sqlinfexp_2$⟩ は，全ての ⟨$sqlinfexp_i$⟩ が SML#で評価される式ならば，SML#で評価される．
6. SML#の関数適用式 ⟨*sqlappexp*⟩ ⟨*sqlatexp*⟩ は SML#で評価される．⟨*sqlappexp*⟩ または ⟨*sqlatexp*⟩ が SML#で評価される式でない場合は構文エラーである．

例えば，SQL クエリ

```
val q = _sql db => select SOME 1 + sum(#a.b) from #db.a group by #a.c;
```

において，部分式 `SOME 1` は SML#で評価され，その値が SQL クエリ断片に埋め込まれる（もし `SOME` が未定義ならば未定義変数エラーが発生する）．`sum(#a.b)` は SML#で評価されず，そのまま SQL クエリ断片に含まれる．`q` を実行したときにサーバーに送信される SQL クエリは以下の通りである．

```
SELECT 1 + SUM(a.b) AS "1" FROM a GROUP BY a.c
```

SML#で評価される式の型は，22.1.1 節で定義した SQL の基本型のうちのいずれかでなければならない．

SQL 評価式に含まれる SML#で評価される式は，通常の SML#の式と同じ順番で，SQL 評価式が評価されるときにただちに評価される．SML#で評価される式に関数適用式を含む SQL 評価式は expansive である．SQL 評価式や SQL クエリは多くの場合多相的であるため，SML#で評価される関数適用式を含む SQL 評価式はトップレベルで value restrcition 警告を引き起こす．例えば，以下のようになる．

```
# _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
```

SQL 評価式をトップレベルに書いたり，多相的に使いたいときは，`fn () =>` ··· を付けるなどして，value restriction を回避する必要がある．例えば，上記の警告は以下のようにすれば回避できる．

```
# fn () => _sql(1 + 2);
val it = fn : ['a, 'b. unit -> ('a -> int, 'b) SQL.exp]
```

22.4.2 SQL 定数式

定数式は全て SML#で評価される式である．定数リテラルの書き方は SML#の文法に準じる．そのため，特に文字列リテラルの書き方については，標準 SQL とは大きく異なる．

`true` および `false` が `SQL.bool3` 型を持たないことに注意しなければならない．これらは第一級の真偽値のリテラルであり，論理演算式としては使えない．論理演算式と真偽値の区別については 22.1.2 節を参照せよ．

SQL99 の機能 T031 に定義されている UNKNOWN リテラルは，SML#では提供されない．これは，主要な RDBMS の中では唯一 T031 を実装している PostgreSQL が UNKNOWN リテラルを持たないからであり，また将来 PostgreSQL 以外に T031 を実装する RDBMS が現れたとしても，UNKNOWN リテラルは NULL リテラルで代用可能だからである（なお，この UNKNOWN の仕様は SQL コアと一貫性が無いことが指摘されている）．

22.4.3 SQL 識別子式

SQL 評価式に現れる全ての ⟨*vid*⟩ および `op` ⟨*longvid*⟩ は，SML#の変数参照である．SQL 評価式の中では，ストラクチャ識別子が一つ以上前置された ⟨*longvid*⟩ には，必ず `op` を前置しなければならない．

SQL 評価式の中ではいくつかの識別子が以下の通り infix 宣言される．

```
infix 7 %
infix 5 like ||
nonfix mod
```

識別子の型と値は，その識別子が SML#で評価される式に現れているかどうかによって異なる．識別子が SML#で評価される式に現れているならば，その型と値は現在の SML#の環境の中でその識別子が束縛された型と値である．そうでなければ，識別子は `SQL.Op` ストラクチャに定義された型と値を持つ．

例えば，式

```
_sql(1 + 2 + #a.b)
```

に現れる 2 つの+は意味が異なる．最初の+は現在の SML#の環境で束縛されている+であり，2 つ目の+は SQL.Op.+である．

SML#で評価される式でない箇所に現れる識別子は，SQL クエリに現れない関数適用式 (⟨*vid*⟩) ⟨*sqlappexp*⟩ の ⟨*vid*⟩ を除いて（22.4.5 節参照），識別子がそのまま大文字に変換されてから SQL クエリに埋め込まれる．SQL.Op ストラクチャの値が実際に参照されるのは，トイプログラムを実行（22.8.3 節参照）した場合に限られる．SQL.Op ストラクチャに定義された識別子の型は SQL 評価式の型付けに用いられる．例えば _sql(1 + #a.b) が SML#で何型を持つかは，SQL.Op.+の型を調べることで分かる．

22.4.4 SQL 関数適用式および SQL 演算子式

SQL 関数適用式および SQL 演算子式は，組み込み構文になっている論理演算子式を除いて，SML#の関数適用式および演算子式と同様に，SML#の式として構文解析される．演算子の結合力は SML#の演算子宣言によって決定される．そのため，infix 宣言の使い方によっては，標準 SQL とは異なる演算子の結合順位で SQL クエリが構築されることに注意が必要である．

標準 SQL とは異なり，SQL 関数適用式の構文は，SML#の関数適用式と同様に関数と引数を空白で区切って書き，引数を丸括弧で囲うことを必要としない．しかし，標準 SQL に似せるために，SML#の式構文が許す範囲で，敢えて引数を丸括弧で囲い，関数と引数を詰めて書くことができる．SELECT 句で集約関数を用いるときなどにこの記法が用いられる．例えば以下のように書ける．

```
fn db => _sql select avg(#e.age)
              from #db.employee as e
              group by ()
```

この avg(#e.age) は，SQL.Op.avg 関数に#e.age を適用する関数適用式である．

論理演算子以外の SQL の演算子および関数は，SQL.Op ストラクチャに定義された SML#のライブラリ関数として提供されている．SML#で評価されない SQL 評価式で使用可能な関数および演算子の一覧は第 22.9.2 節にある．

SQL 関数適用式および SQL 演算子式は SML#のそれらと同様に型付けされる．

22.4.5 型キャスト式

SML#の SQL 評価式では，識別子だけを丸括弧で囲った式は特別な意味を持つ．丸括弧で囲った識別子を式の前に前置すると，その式の評価結果に丸括弧で囲った識別子が指す関数を適用することを表す．ただし，丸括弧で囲った識別子はサーバーに送信される SQL クエリには現れない．丸括弧で囲われた識別子は SQL.Op ストラクチャに定義された識別子でなければならない．以下，丸括弧で囲われた識別子を前置した式を，型キャスト式という．

型キャスト式は，SML#がサポートしない暗黙のキャストやオーバーロードへの対応として，SML#で型の辻褄を合わせるために用いられる．主な使用用途のひとつは option 型や数値型の取り扱いである．22.1 節で述べたとおり，SML#は NULL になる可能性を option 型を用いて型上で区別する．一方，標準 SQL では，NULL はどの基本型にも含まれる．この違いのため，SQL では型エラーでないクエリでも，SML#では型エラーとなることがある．また，数値型についても，標準 SQL では暗黙に数値型のキャストが行われる一方，SML#は暗黙のキャストを行わないため，SML#では型エラーとなることがある．例えば，部署ごとの平均年齢より若い人の問い合わせる以下のクエリを見てみよう．

```
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 ())
```

この式には型がつくものの，employee テーブルの age カラムの型は SQL.numeric option と推論される．従って，age が int 型のデータベースに対してこのクエリを実行できない．この原因は，集約関数 avg の結果とカラム#e.age を比較していることにある．avg の結果の型は SQL.numeric option 型である．SML#における SQL 比較演算子の型付け規則により，比較演算子の左右の式の型は一致していなければならない．従って，#e.age の型は SQL.numeric option 型となる．標準 SQL であれば，暗黙に NUMERIC 型へのキャストが行われるため，age カラムの型は数値型ならばどの型でも良いはずである．

このような，SML#がサポートしない暗黙のキャストやオーバーロードに対応するため，以下のように比較演算の箇所に Num 関数への適用を加えると，任意の数値型の age カラムに対してこのクエリを実行できるようになる．

```
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 ())
```

この (Num) は SQL クエリの生成では無視される．従って (Num) の有無にかかわらずサーバーに送信される SQL クエリは変化しない．(Num) は SQL 評価式の型付けのためだけに静的に評価される．

型キャスト式のために用意されている関数の一覧については 22.9.1 節を参照せよ．

22.4.6 SQL 論理演算式

以下の論理演算式は組み込み構文である．

- not ⟨*sqlexp*⟩
- ⟨$sqlexp_1$⟩ and ⟨$sqlexp_2$⟩
- ⟨$sqlexp_1$⟩ or ⟨$sqlexp_2$⟩
- ⟨*sqlexp*⟩ is (not)? ⟨*sqlis*⟩

これらのうち and は，SML#の他の構文と曖昧にならないように，and を含む式全体が括弧で囲われていなければならない．例えば以下の例は構文エラーとなる．

```
# fn () => _sql where #t.c >= 10 and #t.c <= 20;
(interactive):1.31-1.35 Error: Syntax error: deleting AND HASH
```

以下の例のように and を含む式全体を括弧で囲むことで構文エラーを回避できる．

```
# 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]
```

これらの論理演算式はそれぞれ，その全ての部分式が SQL.bool3 型のとき，SQL.bool3 型を持つ．また，ライブラリ関数として提供される SQL 比較演算子（22.9.2 節参照）もすべて，SQL.bool3 型のクエリを返す．

22.1.2 節で述べたように，標準 SQL での真偽値の取り扱いに関する混乱を避けるため，SML#は論理演算式の型と真偽値の型を区別する．従って，SQL の BOOLEAN 型のカラムの参照や，true などの真偽値リテラルは，論理演算式として書けないことに注意が必要である．例えば，以下の例は型エラーになる．

```
# 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
```

一方，以下の例は型エラーにならない．

```
# fn () => _sql(#t.c = true is false);
val it = fn : ['a, 'b. unit -> ('a -> SQL.bool3, 'b) SQL.exp]
```

なぜなら，`true` は真偽値リテラルであるが，`#t.c = true` は `SQL.bool3` 型の比較演算式だからである．

22.4.7 SQL カラム参照式

あるテーブルのカラムを参照するには，`#`⟨lab_1⟩`.`⟨lab_2⟩ と書く．ここで，⟨lab_1⟩ はテーブルの名前，⟨lab_2⟩ はカラムの名前である．SQL カラム参照式の型は，文脈で与えらえるテーブル集合の型のうち，テーブル ⟨lab_1⟩ のカラム ⟨lab_2⟩ の型である．

標準 SQL では，テーブル名を省略してカラムを参照できるときがあるが，SML#の SQL 評価式では，すべてのカラム名にはテーブル名が前置されていなければならない．また，SML#の他の構文との衝突を避けるため，SQL カラム参照式には`#`を前置する．

テーブル名およびカラム名の大文字小文字は，SML#プログラムの中では区別される．例えば，標準 SQL では

```
SELECT Employee.name FROM EMPLOYEE
```

のように，大文字と小文字を混ぜて `EMPLOYEE` テーブルを参照できるが，SML#では

```
_sql db => select #Employee.name from #db.EMPLOYEE
```

と書くと型エラーとなる．SQL カラム参照式の評価結果には，大文字小文字を変換することなく，SML#プログラムに書いた通りのテーブル名およびカラム名が埋め込まれる．

SQL では名前が付けられていないテーブル（または関係）のカラムを参照することがある．その代表的な例は，SELECT 句が計算したカラムを ORDER BY 句から参照するときである．このようなカラムへの参照は，SML#では`#.`⟨lab⟩ と書く．その型はカラム ⟨lab⟩ の型である．例えば，SQL クエリ

```
SELECT e.name AS name, e.age + 1 AS nextAge
FROM employee AS e
ORDER BY nextAge
```

を，SML#では

```
fn db => _sql select #e.name as name, #e.age + 1 as nextAge
              from employee as e
              order by #.nextAge
```

と書く．

22.4.8 SQL サブクエリ

SML#の SQL 評価式では，以下の 2 種類のサブクエリを書くことができる．

- SELECT サブクエリ：（ ⟨$sqlselect$⟩ ）

- EXISTS サブクエリ： `exists (` ⟨*sqlselect*⟩ `)`

SML#の他の構文との見た目上の統一感を出すために，SQL 予約語の前に`_sql` を書いてもよい．この`_sql` は単純に無視される．

SELECT サブクエリ (⟨*sqlselect*⟩) の ⟨*sqlselect*⟩ は，1 行 1 列を返すクエリでなければならない．このうち，⟨*sqlselect*⟩ が 1 列を返すクエリであることは，SML#の型システムによって静的に検査される．⟨*sqlselect*⟩ は，SML#で

`({1:` τ`} list,` w`) SQL.query`

型を持たなければならない．このとき，サブクエリ (⟨*sqlselect*⟩) の型は，ある τ' について $(\tau', w) \triangleright \tau$ である．サブクエリが返す行数は，クエリを実行するサーバーが実行時にチェックする．クエリを実行したときに ⟨*sqlselect*⟩ が 0 行または 2 行以上の行を返した場合は，例外`SQL.Exec` が発生する．

EXISTS サブクエリ `exists (` ⟨*sqlselect*⟩ `)` は，⟨*sqlselect*⟩ の型が

(`$\tau$ list,` w`) SQL.query`

のとき，ある τ' について型 $(\tau', w) \triangleright$ `SQL.bool3` を持つ．

22.4.9 SQL 評価式の埋め込み

SQL 評価式 ⟨*sqlexp*⟩ を第一級の SML#オブジェクトとして取り出すには`_sql(` ⟨*sqlexp*⟩ `)` と書く．⟨*sqlexp*⟩ の型が $(\tau', w) \triangleright \tau$ のとき，式`_sql(` ⟨*sqlexp*⟩ `)` の型は (`$\tau'$ -> $\tau$,` w`) SQL.exp` である．例えば，

```
val q = _sql(1 + #employee.salary)
```

の型は

```
val q : [ 'a#{employee: 'b}, 'b#{salary: int}, 'w. ('a -> int, 'w) SQL.exp ]
```

である．

このようにして取り出した SQL 評価式の断片を別の SQL 評価式に埋め込むには埋め込み式 (`...` ⟨*exp*⟩) を用いる．例えば，上述の SQL 評価式断片 `q` を用いて以下のように SQL 評価式を構築できる．

```
_sql((...q) > 10)
```

この SQL 評価式は以下の SQL クエリを表す．

```
1 + employee.salary > 10
```

式 (`...` ⟨*exp*⟩) の型は，⟨*exp*⟩ の型が (`$\tau'$ -> $\tau$,` w`) SQL.exp` のとき，$(\tau', w) \triangleright \tau$ である．埋め込み先の式が参照するテーブルの型と τ' に齟齬がある場合，型エラーとなる．例えば，以下の式は型エラーである．

```
_sql((...q) > 10 and #employee.salary = "abc")
```

(`...` ⟨*exp*⟩ 記法に含まれる ⟨*exp*⟩ は，SQL 評価式ではなく，SML#の式である．この ⟨*exp*⟩ の中では，通常の SML#の式と同様に，SQL 予約語は予約語ではなく識別子と解釈される．

SQL 評価式を埋め込んだ結果に定数式が現れたとしても，その定数式は SML#で評価されない．SQL 評価式が SML#で評価される式かどうかは，静的な構文構造によってのみ定まる．例えば，以下の再帰関数 `nat` n は，`1` が n 個ならんだ SQL 評価式 `1 + 1 + ⋯ + 1` を構築する関数であり，その評価結果である n を計算する関数ではない．

```
# 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 クエリ

SML#では，SML#では，SELECT クエリの各句を独立に部分的に定義し，それらを合成することで，SELECT クエリ全体を構築することができる．SELECT クエリを構成する句 ⟨*sqlclause*⟩ には以下のものがある．

⟨*sqlclause*⟩ ::= ⟨*sqlSelectClause*⟩ SELECT 句
| ⟨*sqlFromClause*⟩ FROM 句
| ⟨*sqlWhereClause*⟩ WHERE 句
| ⟨*sqlOrderClause*⟩ ORDER BY 句
| ⟨*sqlOffsetClause*⟩ OFFSET 句
| ⟨*sqlLimitClause*⟩ LIMIT 句

各句の構文の詳細は本節で副節に分けて定義する．

SELECT クエリ全体はこれらの句および GROUP BY 句 ⟨*sqlGroupClause*⟩ を組み合わせて構成される．その構文は以下の通りである．

⟨*sqlselect*⟩ ::= ⟨*sqlSelectClause*⟩ SELECT 句を持つ SELECT クエリ
⟨*sqlFromClause*⟩
(⟨*sqlWhereClause*⟩)?
(⟨*sqlGroupClause*⟩)?
(⟨*sqlOrderClause*⟩)?
(⟨*sqlLimitClause*⟩)?
| `select ...` `(` ⟨*exp*⟩ `)` SELECT 句が埋め込まれる SELECT クエリ
⟨*sqlFromClause*⟩
(⟨*sqlWhereClause*⟩)?
(⟨*sqlGroupClause*⟩)?
(⟨*sqlOrderClause*⟩)?
(⟨*sqlLimitClause*⟩)?
| `select ...` `(` ⟨*exp*⟩ `)` SELECT クエリ全体の埋め込み

最初の 2 つの構文が SELECT クエリを構成する．SELECT クエリは，SELECT 句と FROM 句を必ず含む．その他の句は任意である．一部の RDBMS の実装（例えば PostgreSQL など）では FROM 句を持たない SELECT クエリが許されているが，SML#では SELECT クエリは必ず FROM 句を持たなければならない．2 つ目の構文冒頭の `select ... (` ⟨*exp*⟩ `)` については後述する．

SELECT クエリを構成する各句が以下の型を持つならば，SELECT クエリ全体は (τ, w) `SQL.query` 型を持つ．

- ⟨*sqlFromClause*⟩ は (τ_1, w) `SQL.from` 型を持つ．
- ⟨*sqlWhereClause*⟩ は存在するならば $(\tau_1$ `->` $\tau_1, w)$ `SQL.whr` 型を持つ．
- GROUP BY 句が存在するとき，τ_1 をグループ化した型 τ_2 が計算される（22.5.4 節参照）．GROUP BY 句がなければ，$\tau_2 = \tau_1$ である．
- ⟨*sqlSelectClause*⟩ は，(τ_2, τ, w) `SQL.select` 型を持つ．
- ⟨*sqlOrderClause*⟩ は存在するならば $(\tau$ `->` $\tau, w)$ `SQL.orderby` 型を持つ．
- ⟨*sqlLimitClause*⟩ は存在するならば $(\tau$ `->` $\tau, w)$ `SQL.limit` 型を持つ．

例えば，以下は完全な SELECT クエリの例である．

```
val q = fn db => _sql select #t.name as name, #t.age as age
                      from #db.employee as t
                      where #t.age >= 20
```

3 つ目の構文 `select...(`⟨*exp*⟩`)` は，SELECT クエリを直接書く代わりに，SML#の式 ⟨*exp*⟩ を評価した結果得られた SELECT クエリを埋め込む．⟨*exp*⟩ は `SQL.query` 型でなければならない．例えば，以下は上述の例のクエリ `q` をサブクエリとして他のクエリ `q2` に埋め込む例である．

```
val q2 = fn db => _sql select #t.name as name
                       from (select...(q db)) as t
                       where #t.name like "%Taro%"
```

2 つ目の構文および以下の副節に示す各句の構文の定義のとおり，一部の例外を除き，句名に続けて `...(`⟨*exp*⟩`)` と書くことで，SML#の式を評価した結果をその句とすることができる．例えば，上述の例と同等の SELECT クエリを，以下のように，SELECT 句と FROM 句を独立に定義したのち，`...(`⟨*exp*⟩`)` 記法を用いて組み合わせることで構築することができる．

```
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)
```

これら `...(`⟨*exp*⟩`)` 記法に含まれる ⟨*exp*⟩ は，SQL 評価式ではなく，SML#の式である．この ⟨*exp*⟩ の中では，通常の SML#の式と同様に，SQL 予約語は予約語ではなく識別子と解釈される．

22.5.1 SELECT 句

SELECT 句 ⟨*sqlSelectClause*⟩ の構文は以下の通りである．

⟨*sqlSelectClause*⟩ ::= `select` (⟨*distinct_all*⟩)? ⟨*sqlSelectField*⟩ , ⋯ , ⟨*sqlSelectField*⟩
⟨*distinct_all*⟩ ::= `distinct` | `all`
⟨*sqlSelectField*⟩ ::= ⟨*sqlexp*⟩ (`as` ⟨*lab*⟩)?

SELECT 句は 1 つ以上のフィールド ⟨*sqlSelectField*⟩ からなる．各フィールドはラベル ⟨*lab*⟩ を持つ．i 番目のフィールド（最初のフィールドを 1 とする）の `as` ⟨lab_i⟩ が省略されている場合，`as` i が指定されているとみなす．どのフィールドのラベルも他のフィールドのラベルと異なっていなければならない．

n 個のフィールドを持つ ⟨*sqlSelectClause*⟩ の型は，その i 番目 ($1 \leq i \leq n$) のフィールド ⟨$sqlexp_i$⟩ `as` ⟨lab_i⟩ の SQL 評価式 ⟨$sqlexp_i$⟩ の型が $(\tau, w) \triangleright \tau_i$ のとき，

`(`τ`, {`⟨lab_1⟩ `:` τ_1`, ...,` ⟨lab_n⟩ `:` τ_n`} list,` w`) SQL.query`

である．

22.5.2 FROM 句

FROM 句 ⟨*sqlFromClause*⟩ の構文は以下の通りである．

⟨*sqlFromClause*⟩ ::= `from` ⟨*sqlTable*⟩ , ⋯ , ⟨*sqlTable*⟩
| `from ... (`⟨*exp*⟩`)`

最初の構文が FROM 句を構築する．2 つ目の構文は式 ⟨*exp*⟩ の評価結果を FROM 句として埋め込む．⟨*exp*⟩ は `SQL.from` 型でなければならない．

最初の構文が含むコンマは，SML#の組式のコンマと曖昧になることがある．例えば，

```
(1, _sql from #db.t1, #db.t2, #db.t3)
```

は，1 と FROM 句のペアなのか，あるいは 2 番目の要素を `_sql from #db.t1` とする 4 つ組なのか，曖昧である．前者を意図しているならば

```
(1, _sql (from #db.t1, #db.t2, #db.t3))
```

後者を意図しているならば

```
(1, _sql (from #db.t1), #db.t2, #db.t3)
```

のように，括弧を付けて書かなければならない．22.2 節で示した_sql 式が書ける位置の制限は，この曖昧さを回避するために導入されている．なお，

```
(_sql from #db.t1, #db.t2, #db.t3)
```

は，開き括弧の直後に_sql が来ているため，全体がひとつの SQL 構文であると認識される．

⟨*sqlTable*⟩ はテーブルを表す式であり，その構文は以下の通りである．

⟨*sqlTable*⟩	::=	`#` ⟨*vid*⟩ `.` ⟨*lab*⟩	テーブル参照
	|	⟨*sqlTable*⟩ `as` ⟨*lab*⟩	テーブルのラベル付け
	|	`(` (`_sql`)? ⟨*sqlselect*⟩ `)`	テーブルサブクエリ
	|	⟨*sqlTable*⟩ (`inner`)? `join` ⟨*sqlTable*⟩ `on` ⟨*sqlexp*⟩	テーブルの内部結合
	|	⟨*sqlTable*⟩ `cross join` ⟨*sqlTable*⟩	テーブルの直積
	|	⟨*sqlTable*⟩ `natural join` ⟨*sqlTable*⟩	テーブルの自然結合
	|	`(` ⟨*sqlTable*⟩ `)`	

ラベル付け構文について以下の規則がある．

- ラベル付け構文は他のどの ⟨*sqlTable*⟩ 構文よりも結合力が強い．
- テーブル参照`#` ⟨*vid*⟩ `.` ⟨*lab*⟩ に直接かかるラベル付け構文が無い場合，`as` ⟨*lab*⟩ が補われ，テーブルと同名のラベルが付けられているものとみなす．

また，⟨*sqlTable*⟩ の構文には以下の制限がある．

- ⟨*sqlFromClause*⟩ に現れる全ての `as` ⟨*lab*⟩ は互いに異なっていなければならない．従って，同じ名前のテーブルを 2 度以上参照するときは，それぞれの参照に別のラベルを `as` で付けなければならない．
- ⟨$sqlTable_1$⟩ `natural join` ⟨$sqlTable_2$⟩ の ⟨$sqlTable_1$⟩ および ⟨$sqlTable_2$⟩ は，内部結合または直積であってはならない．
- ⟨*sqlTable*⟩ `as` ⟨*lab*⟩ の ⟨*sqlTable*⟩ は内部結合および直積であってはならない．

n 個の ⟨$sqlTable_i$⟩ `as` ⟨lab_i⟩ ($1 \leq i \leq n$) を持つ FROM 句の型は，⟨$sqlTable_i$⟩ の型を τ_i とするとき，

`({`⟨lab_1⟩ `:` τ_1`,` ...`,` ⟨lab_n⟩ `:` τ_n`} list,` w`) SQL.from`

である．

⟨$sqlTable_i$⟩ の型は以下の通りである．

- `#` ⟨*vid*⟩ `.` ⟨*lab*⟩ の型は，SML#の変数 ⟨*vid*⟩ の型が `(`τ`,` w`) SQL.db` でかつ τ がフィールド ⟨*lab*⟩`:`τ' を持つレコード型のとき，τ' である．
- ⟨*sqlTable*⟩ `as` ⟨*lab*⟩ の型は ⟨*sqlTable*⟩ の型に等しい．
- `(` ⟨*sqlselect*⟩ `)` の型は，⟨*sqlselect*⟩ の型が `(`τ`,`w`) SQL.query` のとき τ である．
- ⟨$sqlTable_1$⟩ `natural join` ⟨$sqlTable_2$⟩ の型は，2 つのテーブルを自然結合したレコード型である．

SML#は内部結合式および直積式の型を計算しない．そのため，内部結合および直積の結果に `as` で直接にラベル付けをすることはできない（構文上制限されている）．内部結合および直積の結果の型は FROM 句全体の型であるレコード型として現れる．

FROM 句は，概念上，テーブルを結合したひとつのテーブルを計算する．FROM 句の型に現れるレコード型`{`⟨lab_1⟩ `:` τ_1`,` ...`,` ⟨lab_n⟩ `:` τ_n`}`は，このテーブルの各行における，ラベル付けされた成分を表している．他の句に現れる SQL カラム参照式`#` ⟨lab_1⟩ `.` ⟨lab_2⟩ の ⟨lab_1⟩ は，各成分に付けられたこれらのラベルを指す．

22.5.3 WHERE 句

WHERE 句 ⟨*sqlWhereClause*⟩ の構文は以下の通りである．

⟨*sqlWhereClause*⟩ ::= `where` ⟨*sqlexp*⟩
| `where ... (` ⟨*exp*⟩ `)`

最初の構文が WHERE 句を構成する．⟨*sqlexp*⟩ の型が $(\tau, w) \triangleright$ `SQL.bool3` のとき，WHERE 句の型は `(`τ `->` τ`,` w`)` `SQL.whr` である．

2 つ目の構文は式 ⟨*exp*⟩ の評価結果を WHERE 句として埋め込む．⟨*exp*⟩ は `SQL.whr` 型でなければならない．

22.5.4 GROUP BY 句

GROUP BY 句 ⟨*sqlGroupClause*⟩ の構文は以下の通りである．

⟨*sqlGroupClause*⟩ ::= `group by` ⟨*sqlexp*⟩ , ⋯ , ⟨*sqlexp*⟩ (`having` ⟨*sqlexp*⟩)?
| `group by ()`

他の句とは異なり，GROUP BY 句には `...(`⟨*exp*⟩`)` 記法が存在せず，従って SELECT クエリから分離して構築することができない．GROUP BY 句は，構文上，ひとつの SELECT クエリの一部として SELECT 句および FROM 句と共に現れる．

FROM 句の型を `(`τ`,` w`)` `SQL.from` 型とすると，GROUP BY 句にコンマ区切りで並べられている ⟨$sqlexp_i$⟩ はそれぞれ，$(\tau, w) \triangleright \tau_i$ 型でなければならない．GROUP BY 句はこれらの式の評価結果の組をキーとして，FROM 句が計算したテーブルを複数の行のグループに分割する．行のグループのテーブルの型 τ' は後述する方法で計算される．

GROUP BY 句は高々ひとつの HAVING 句を持ってもよい．HAVING 句の SQL 評価式は GROUP BY 句の評価後に行のグループをフィルタするための条件式であり，従ってその型は $(\tau', w) \triangleright$ `SQL.bool3` である．

2 つ目の構文 `group by ()` は，FROM 句が計算したテーブル全体をひとつのグループとすることを表す標準 SQL の構文である．伝統的な標準 SQL では，GROUP BY 句を書かずに SELECT 句で集約関数を用いると，そのクエリはテーブル全体を集約するものとみなされる．例えば，

```
SELECT avg(e.age) FROM employee AS e
```

は，テーブル全体の `e.age` の平均を求める正しい SQL クエリである．一方，SML#では，テーブル全体を集約するクエリには `group by ()` を書かなければならない．上述の SQL クエリは，SML#では以下のように書く．

```
fn db => _sql select avg(#e.age) from #db.employee as e group by ()
```

データベースサーバーには，SML#プログラムに書かれている通り，

```
SELECT avg(e.age) FROM employee AS e GROUP BY ()
```

が送信される．

GROUP BY 句が計算する行のグループの型 τ' は，おおよそ以下のようにして計算される．

1. グループ化する前の行の型を
$$\tau = \{k_1 : \tau_1, \ldots, k_n : \tau_n\}$$
とする．FROM 句が出力するテーブルの型は τ `list` である．

2. 行をグループ化すると，その型は τ `list list` となる．

3. 行のグループ $\{k_1 : \tau_1, \ldots, k_n : \tau_n\}$ `list` を転置し，$\tau^t = \{k_1 : \tau_1$ `list`$, \ldots, k_n : \tau_n$ `list`$\}$ とする．

手順 3 の転置を行うためには，カラム名の集合 $k_1, \ldots, k_n$ が静的に確定している必要がある．SML#は，カラム名の集合を，同じクエリに構文上含まれるカラム参照式の集合から取得する．GROUP BY 句にキーとして指定されているか，SELECT 句などに含まれるグループを指すカラム参照式#$\langle lab_1 \rangle$. $\langle lab_2 \rangle$ それぞれについて，もしそのカラム参照式が GROUP BY 句に書かれたキーのいずれかひとつに一致するならば，そのカラムを単一の値のカラムとして GROUP BY 句の結果の型に含める．そうでないならば，そのカラムは値のリストを持つカラムとなる．参照されないカラムの型は，GROUP BY 句の出力の型として計算されない．

行の集合の型の計算は，あくまで構文上の文脈に基づいて行われ，変数の参照関係などを考慮しない．例えば，

```
val q = fn db => _sql select #e.department, avg(#e.salary)
                      from #db.employee as e
                      group by #e.department
```

は GROUP BY 句を持つ正しいクエリである一方，SELECT 句を分離した以下の例では型エラーが発生する．

```
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
```

なぜなら，SELECT 句が GROUP BY 句と構文上同じ SELECT クエリに現れる最初の例では，GROUP BY 句の出力の型に`#e.department` と`#e.salary` の両方が適切に含まれる一方，2 つ目の例では SELECT 句が GROUP BY 句を持つ SELECT クエリの中に無いため，GROUP BY 句の結果は全く参照されないものとして GROUP BY 句の型が計算されるからである．良い習慣として，`group by` を含むクエリを書くときは，`select` に`...(`$\langle exp \rangle$`)` 記法を使わないことが望ましい．

以下は，注意深い読者のための補足説明である．SELECT 句を分離しても，もしその SELECT 句が GROUP BY 句で指定したキーしか参照しないならば，型エラーは発生しない．例えば上述の例の `s` の定義を書き換えて `avg(#e.salary)` を削除し，

```
val s = _sql select #e.department
val q = fn db => _sql select...(s)
                      from #db.employee as e
                      group by #e.department
```

としたならば，型エラーは発生しない．なぜなら，`#e.department` は GROUP BY 句で指定されたキーであり，従って SELECT 句での参照の有無にかかわらず GROUP BY 句の型に含まれるからである．

22.5.5 ORDER BY 句

ORDER BY 句 $\langle sqlOrderClause \rangle$ の構文は以下の通りである．

$\langle sqlOrderClause \rangle$	::=	`order by` $\langle sqlOrderKey \rangle$, $\cdots$ $\langle sqlOrderKey \rangle$
	|	`order by ...` ($\langle exp \rangle$)
$\langle sqlOrderKey \rangle$	::=	$\langle sqlexp \rangle$ ($\langle asc\_desc \rangle$)?
$\langle asc\_desc \rangle$	::=	`asc` | `desc`

最初の構文が ORDER BY 句を構築する．2 つ目の構文は式 $\langle exp \rangle$ の評価結果を ORDER BY 句として埋め込む．$\langle exp \rangle$ は `SQL.orderby` 型でなければならない．

ORDER BY 句は SELECT 句が計算した結果のテーブルに対して行の並び替えを行う．SELECT 句の結果の型を τ とすると，ORDER BY 句に現れる各 $\langle sqlexp_i \rangle$ は $(\tau, w) \triangleright \tau_i$ 型であり，ORDER BY 句全体の型は (`'`τ `->` τ, w) `SQL.orderby` である．

ORDER BY 句には，後方互換性を持たない仕様変更や，ベンダー独自の拡張が存在する．SML#でORDER BY 句にキーとして書けるのは，SELECT 句が出力する結果のカラムを参照する任意の式である．ORDER BY 句から SELECT 句の結果のカラムを参照するには，テーブル名を持たないカラム参照式 `#.`⟨*lab*⟩ を用いる．例えば以下のように書く．

```
fn db => _sql select #e.name as name, #.age as age
              from #db.employee as e
              order by #.age
```

多くのデータベースエンジンは，SELECT 句の結果に含まれないカラムをキーとしてソートすることを許している一方，そのようなカラム参照は SML#では型エラーとなる．例えば，以下のクエリは型エラーである．

```
fn db => _sql select #e.name as name, #.age as age
              from #db.employee as e
              order by #e.department
```

22.5.6 OFFSET 句または LIMIT 句

OFFSET 句および LIMIT 句は，SELECT 句の計算結果から指定範囲の行のみを取り出す．標準 SQL に定められているのが OFFSET 句，ベンダーによる拡張が LIMIT 句であり，これらは機能的に等価である．どちらも広く受け入れられているため，SML#ではこれら両方をサポートする．

⟨*sqlOffsetOrLimitClause*⟩ の構文を以下に示す．

⟨*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*⟩

予約語 `offset` の使い方が LIMIT 句および OFFSET 句でそれぞれ異なることに注意が必要である．LIMIT 句は OFFSET 副句を持ち，予約語 `limit` から始まる．OFFSET 句は FETCH 副句を持ち，予約語 `offset` から始まる．これらの副句を混ぜて使うことはできない．また，OFFSET 句に定数でない式を書く場合は，式が括弧で囲まれていなければならない．

これらの句に現れる ⟨*sqlexp*⟩ または ⟨*sqlatexp*⟩ の型は $(\{\}, w) \triangleright$ `int` である．従って，これらの句の中にカラム参照式を書くことはできない．

いくつかのデータベースエンジンでは，1 つのクエリの中でこれらの句を複数指定したり，句の順序を入れ替えたりすることを許している．SML#では，標準 SQL に従い，これらの副句は主句の後に続いて現れなければならない．また，OFFSET 句および LIMIT 句はどちらも高々1 つまでしか書けない．

22.5.7 相関サブクエリ

SELECT クエリがネストしているとき，内側の SELECT クエリは外側のクエリのサブクエリである．相関サブクエリとは，外側のクエリの FROM 句が導入するカラムを参照するサブクエリを言う．以下は，標準 SQL で書かれた相関サブクエリの例である．

```
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 ())
```

SML#では，サブクエリを SELECT 句など ⟨*sqlexp*⟩ が書ける場所（22.4.8 節参照）と FROM 句（22.5.2 節参照）に書くことができる．サブクエリは，以下の構文上の制約を満たすとき，相関サブクエリであってもよい．

1. そのサブクエリおよび構文上そのサブクエリを囲む全ての SELECT クエリの FROM 句が `from...(`⟨*exp*⟩`)` の形でないとき（22.5 節参照），そのサブクエリは相関サブクエリであってもよい．

以下は，上述の標準 SQL で書いた相関サブクエリを SML#で書いた例である．

```
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 ())
```

ネストした SELECT クエリのいずれかの FROM 句が `from...(`⟨*exp*⟩`)` の場合，サブクエリは相関サブクエリと解釈されない．例えば以下のように，上述の例のサブクエリの FROM 句を `from...(`⟨*exp*⟩`)` に書き直したとき，

```
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
```

サブクエリの`#e.department` の `e` は，外側の `from #db.employee as e` の `e` ではなく，`from ...(x db)` が導入する `e` を参照する，と解釈される．従って，`x db` は `e` を束縛する FROM 句でなくてはならず，この例は型エラーとなる．外側の FROM 句を `from...(`⟨*exp*⟩`)` にした場合も，

```
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 ())
```

サブクエリは相関サブクエリとみなされず，サブクエリの`#e.department` の `e` は未定義となり，型エラーとなる．

GROUP BY 句と相関サブクエリは，期待される通りに組み合わせることができる．例えば，以下は，各部署の平均給料を上回る給料をもらっている最年少の人の年齢を問い合わせるクエリである．

```
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
```

このクエリでは，外側の GROUP BY 句でグループ化したカラム`#e.salary`を，サブクエリの中で`min`関数を用いて集約している．GROUP BY 句の型は構文上の文脈から計算されるが（22.5.4 節参照），SML#コンパイラはグループが相関サブクエリからも参照されていることを正しく認識する．

以下は，注意深い読者のための補足説明である．相関サブクエリは，構文上，他のクエリの内側になければならないが，だからといって，相関サブクエリ式を評価した結果得られる SQL 評価式が，ML の静的スコープのように構文上ネストするクエリとだけ相関する，というわけではない．相関サブクエリが書ける場所の制約は，あくまで相関サブクエリの型を計算できるようにするための規則に過ぎない．相関サブクエリを含む SQL 評価式が第一級市民である以上，SML#の言語機能を駆使して，あるクエリ A の内側で作った相関サブクエリ B を取り出し別のクエリ C に埋め込むことも，型さえ合っているならば可能である．このとき，C に埋め込まれた B が参照する外側のテーブルは，A のものではなく，C のものである．SQL クエリがどのような経緯で組み立てられたとしても，その組み立て操作の型が正しいならば，結果として作られる SQL クエリは正しい．

22.6 SQL コマンド

SML#は，SELECT クエリに加え，以下の規則で定義される SQL コマンドのサブセット ⟨*sqlcommand*⟩ を受け付ける．

⟨*sqlcommand*⟩	::=	⟨*sqlinsert*⟩	INSERT コマンド
	|	⟨*sqlupdate*⟩	UPDATE コマンド
	|	⟨*sqldelete*⟩	DELETE コマンド
	|	⟨*sqltransaction*⟩	トランザクション関連コマンド
	|	(⟨*sqlfn*⟩ ; ⋯; ⟨*sqlfn*⟩)	コマンドのシーケンス
	|	...(⟨*exp*⟩)	SQL コマンドの埋め込み

INSERT，UPDATE，DELETE，トランザクション関連コマンドの型は，

```
(unit, w) SQL.command
```

である．各コマンドの詳細は以下の各副節で与える．

(⟨$sqlfn_1$⟩ ; ⋯; ⟨$sqlfn_n$⟩)（n は 2 以上でなければならない）は，一度に送信されるセミコロンで区切られた一連のコマンドを表す．その型は最後のコマンド ⟨$sqlfn_n$⟩ の型に等しい．

...(⟨*exp*⟩) は，SML#の式を評価した結果をコマンドとして埋め込む．

22.6.1 INSERT コマンド

INSERT コマンドの構文は以下の通りである．

⟨*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*⟩

以下の構文上の制約がある.

- `(`⟨*lab*⟩`,` ⋯`,` ⟨*lab*⟩`)` に現れるラベルは全て互いに異なっていなければならない.
- ⟨*insertRow*⟩ に現れる ⟨*insertVal*⟩ の数は，`(`⟨*lab*⟩`,` ⋯`,` ⟨*lab*⟩`)` のラベルの数と一致しなければならない.

INSERT コマンドは，`values` 句で指定された各行あるいは ⟨*sqlselect*⟩ クエリの評価結果の各行を，指定したテーブルに挿入する．`values` 句に SML#の変数 ⟨*vid*⟩ または `op` ⟨*longvid*⟩ を書いた場合，ラベル $\langle lab_1\rangle, \cdots, \langle lab_n\rangle$ を持つレコードのリストをテーブルに挿入する．挿入しようとしている行が挿入先テーブルのカラムを網羅していないとき，指定のないカラムには，テーブル作成時に指定されたデフォルト値が挿入される．また，`values` 句に `default` と書かれている箇所についてもデフォルト値が挿入される．カラムにデフォルト値が設定されていないときは，コマンドを評価するサーバーで実行時エラーが発生し，`SQL.Exec` 例外が発生する.

型の制約は以下の通りである.

- ⟨*vid*⟩ はテーブル名 ⟨*lab*⟩ を含む `SQL.db` 型でなければならない.
- `(`⟨*lab*$_1$⟩`,` ⋯`,` ⟨*lab*$_n$⟩`)` が書かれているとき，`#`⟨*vid*⟩`.`⟨*lab*⟩ が指示するテーブルおよび ⟨*sqlselect*⟩ は，少なくともカラム $\langle lab_1\rangle, \ldots, \langle lab_n\rangle$ を持っていなければならない．それ以外のカラムを持っていても良い．また，挿入先テーブルと ⟨*sqlselect*⟩ クエリのカラム集合が一致していなくてもよい.
- `(`⟨*lab*$_1$⟩`,` ⋯`,` ⟨*lab*$_n$⟩`)` が書かれていないとき，挿入先テーブルと ⟨*sqlselect*⟩ クエリのカラム集合は一致しなければならない.
- `values` 句に並ぶ各組の i 番目の式 $\langle sqlexp_i\rangle$ の型は，挿入先テーブルのカラム $\langle lab_i\rangle$ の型が τ_i のとき，$(\{\}, w) \triangleright \tau_i$ 型でなければならない.

22.6.2 UPDATE コマンド

UPDATE コマンドの構文は以下の通りである.

⟨*sqlupdate*⟩ ::= `update #`⟨*vid*⟩`.`⟨*lab*⟩
`set` ⟨*updateRow*⟩`,` ⋯`,` ⟨*updateRow*⟩
`(` ⟨*sqlWhereClause*⟩ `)`?
⟨*updateRow*⟩ ::= ⟨*lab*⟩ `=` ⟨*sqlexp*⟩

以下の構文上の制約がある.

- ⟨*updateRow*⟩ の ⟨*lab*⟩ は全て互いに異なっていなければならない.

UPDATE コマンドは，指定したテーブルのうち WHERE 句の条件を満たす行を，SET 句の値で更新する．SET 句で指定されていないカラムは更新されない．新しい値を計算する式には，更新前の行の値を参照するために，⟨*lab*⟩ をテーブル名とするカラム参照式を書いても良い.

型の制約は以下の通りである.

- ⟨*vid*⟩ はテーブル名 ⟨*lab*⟩ を含む SQL.db 型でなければならない.
- 各 ⟨$updateRow_i$⟩ で指定されるカラム ⟨lab_i⟩ はすべて更新先テーブルに含まれていなければならない. 更新先テーブルはそれ以外のカラムを持っていても良い.
- 各 ⟨$updateRow_i$⟩ の式 ⟨$sqlexp_i$⟩ の型は, 更新先テーブルの行の型が τ, カラム ⟨lab_i⟩ の型が τ_i のとき, $(\{\langle lab\rangle : \tau\}, w) \triangleright \tau_i$ 型でなければならない.
- ⟨*sqlWhereClause*⟩ は, 存在するならば, 更新先テーブルの型が τ のとき, (τ -> τ, w) SQL.whr 型でなければならない.

22.6.3 DELETE コマンド

DELETE コマンドの構文は以下の通りである.

⟨*sqldelete*⟩ ::= `delete from #`⟨*vid*⟩`.`⟨*lab*⟩ (`where` ⟨*sqlWhereClause*⟩)?

DELETE コマンドは指定したテーブルのうち WHERE 句の条件を満たす行を削除する.
型の制約は以下の通りである.

- ⟨*vid*⟩ はテーブル名 ⟨*lab*⟩ を含む SQL.db 型でなければならない.
- ⟨*sqlWhereClause*⟩ は, 存在するならば, 更新先テーブルの型が τ のとき, (τ -> τ, w) SQL.whr 型でなければならない.

22.6.4 BEGIN, COMMIT, ROLLBACK コマンド

SML#は, トランザクションを制御するコマンドのうち, 以下のコマンドをサポートする.

⟨*sqltransaction*⟩ ::= `begin` トランザクションの開始
| `commit` トランザクションの終了
| `rollback` トランザクションの中断

22.7 SQL 実行関数式

SQL 実行関数式 `_sql` ⟨*pat*⟩ `=>` ⟨*sqlfn*⟩ は, SQL コマンド ⟨*sqlfn*⟩ をデータベースサーバーで実行する関数を生成する. データベースサーバーへの接続ハンドルを引数としてこの関数を呼び出すと, 式 ⟨*sqlfn*⟩ が評価され, その結果得られた SQL コマンドがサーバーに送信され実行される. 実行に成功したならば, この関数はその実行結果を返し, 実行に失敗したときは SQL.Exec 例外を発生させる. 具体的には, この関数は以下のことを行う.

1. τ SQL.conn 型のデータベースサーバー接続ハンドルを引数として受け取る.
2. データベースサーバー接続ハンドルから (τ, w) SQL.db 型のデータベースの実体を取り出し, パターン ⟨*pat*⟩ の識別子をそれに束縛する.
3. 式 ⟨*sqlfn*⟩ を評価し, (τ', w) SQL.command 型のコマンドを得る. ただし, ⟨*sqlfn*⟩ が ⟨*sqlselect*⟩ の場合は, (τ, w) SQL.query 型の SELECT クエリを (τ SQL.cursor, w) SQL.command 型の SQL コマンドと解釈する.
4. コマンドをサーバーに送信し実行する.
5. 実行に成功したならば, τ' 型のコマンド実行結果を返す.

この振る舞いから分かるように，コマンドの型が (τ, w) SQL.db -> (τ′, w) SQL.command のとき，この関数の型は τ SQL.conn -> τ′ である．一般に τ′ は，⟨sqlfn⟩ が ⟨sqlselect⟩ のとき SQL.cursor 型，そうでないとき unit 型になる．

SELECT クエリや SQL コマンドを SQL 実行関数式に直接書くのが，もっとも簡便な SQL コマンドの実行方法である．例えば，

```
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
```

と書くと，このクエリをデータベースサーバーで実行する関数 q が直ちに得られる．段階的に構築した SELECT クエリを実行可能にするには，以下のように SQL 実行関数内で select...(⟨exp⟩) 記法を用いる．

```
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)
```

SQL コマンドの場合は，SQL 実行関数内に SQL.command 型の式を書く．

```
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)
```

SQL 実行関数式_sql ⟨pat⟩ => ⟨sqlfn⟩ には，以下の型に関する制約がある．

- ⟨sqlfn⟩ の型 (τ,w) SQL.command または (τ′,w) SQL.query の w は，⟨sqlfn⟩ の型以外のどこにも使われていない型変数でなければならない．

この型制約は，複数のデータベースにまたがるクエリの実行を禁止するために導入されている．例えば，以下の関数は型エラーになる．

```
# 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
```

この直接の原因は，select の中に関数 f の引数 exp が入っているために，select ··· の型 (τ,w) SQL.query の w が引数 exp の型 (τ′,w) SQL.exp の w としても用いられており，上述の制約に違反することである．より実践的には，引数 exp は，_sql db => ··· の db とは異なるデータベースを参照する SQL 評価式かもしれないからである．このことを理解するために，例えば，この関数 f を呼び出す別の関数を考えてみよう．

```
fun badExample conn1 conn2 =
    (_sql db2 =>
          select...((f _sql((select #t1.c1 from #db2.t1)) conn1;
                    _sql(select #t2.c2 from #db2.t2))))
      conn2
```

この badExample 関数自体は，f が多相関数ならば，上述の制約にかかわらず型が付く．badExample 関数は，異なる 2 つのデータベース接続ハンドル conn1，conn2 を受け取り，関数 f のクエリを conn1 で，それとは別のクエリを conn2 で実行する．この関数の奇妙なところは，SQL 実行関数の本体から関数 f のクエリを実行しようとしていることである．しかも，関数 f の引数に渡されるのは，conn2 のデータベース db2 を用いて書かれたサブクエリである．結果として，f が実行しようとするクエリは，f が束縛する db と badExample が束縛する db2 の 2 つのデータベースを参照するクエリになる．このようなクエリを実行することはできない．

この制約は，実質的に，プログラム中で SQL 実行関数が書ける場所を制限する．特に注意が必要なのは，上述の関数 f のように，引数として受け取ったクエリを実行する関数は書けないことである．この制限を回避するひとつの方法は，クエリを作る関数と実行する関数を分けることである．例えば，上述の f を SQL 実行関数ではなくクエリを返す以下のような関数に書き直せば，この制限を回避できる．

```
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)
```

SQL コマンドの評価でエラーが発生し SQL.Exec 例外が発生する場合には，以下の場合が含まれる．

1. NOT NULL 制約以外の制約違反．
2. 整数や文字列のオーバーフロー．
3. 0 除算．
4. SML#の SQL 構文では許されているが，クエリを実行するデータベースサーバーが解釈できない構文が使われている．

このうち 4. に関して補足する．SML#がサポートする SQL 構文は，標準規格 SQL99 を基礎とし，主要なベンダーによる共通の拡張と，言語拡張を単純にするための（関数型言語の観点から見て）自然な拡張からなる．標準 SQL に対する準拠度や，クエリの書き方に関する細かな規則は，接続先のデータベースエンジンによって異なる．そのため，SML#の SQL 機能を使用する場合は，SML#が提供する全ての SQL 機能を無条件に使うのではなく，使用するデータベースエンジンに合わせて機能を選択するべきである．例えば，以下は本マニュアル執筆時点で判明している SML#とデータベースの差異である．

- PostgreSQL では，FROM 句において，テーブルを結合した結果に AS で名前が付けられているとき，結合対象のテーブルに AS で付けた名前を SELECT 句から参照することができない．例えば，以下のクエリはテーブル x が参照できないとしてエラーとなる．

  ```
  SELECT x.col FROM (a AS x NATURAL JOIN b AS y) AS z
  ```

- SQLite3 は group by () 記法をサポートしない．代わりに group by ""などを用いることができる．

22.8 SQL ライブラリ：SQL ストラクチャ

SML#は，SQL 関連の拡張構文と共に使用するための関数や型をライブラリとして提供する．SQL に関連する全ての型と関数は，インターフェースファイル sql.smi がプロバイドする SQL ストラクチャに含まれる．SQL 機能を使用して書かれた SML#のソースファイルは，そのインターフェースファイルに以下の一行を加え，sql.smi を参照しなければならない．

```
_require "sql.smi"
```

SQL ストラクチャのシグネチャは以下の通りである．

```
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 : （22.8.1 節参照）
  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 : （22.9 節参照）
  Structure Numeric : （22.10 節参照）
  Structure Decimal = Numeric
end
```

以下，22.1 節で定義した型を除き，これらの定義を役割ごとに副節に分けて説明する．

22.8.1　データベースサーバーへの接続

- `exception Connect of string`
  データベースへの接続に関するエラーを表す例外．`string` はエラーメッセージである．
- `exception Link of string`

データベーススキーマの型チェックに関するエラーを表す例外．string はエラーメッセージである．

- `type backend`

  _sqlserver 構文に書く接続先情報の型（22.3 節参照）．以下の関数のいずれかを用いてこの型を持つ式を書く．

- `val postgresql : string -> backend`

  SQL.postgresql *param* は，PostgreSQL サーバーへの接続情報を返す．文字列 *param* は，PostgreSQL ライブラリ libpq の接続文字列である．接続文字列の詳細は PostgreSQL マニュアルを参照せよ．*param* が適切でない場合は SQL.Connect 例外が発生する．

  サポートする PostgreSQL の型と SML#の型との対応は以下の通りである．

PostgreSQL	SML#
INT, INT4	int
BOOLEAN	bool
TEXT, VARCHAR	string
FLOAT8	real
FLOAT4	real32

- `val mysql : string -> backend`

  SQL.mysql *param* は，MySQL サーバーへの接続情報を返す．文字列 *param* には，“キー=値” という形式の指定を空白文字で区切って並べる．指定可能なキーとその意味は以下の通りである．

キー	説明
host	MySQL サーバーのホスト名
port	MySQL サーバーのポート番号
user	MySQL サーバーにログインするときのユーザー名
password	MySQL サーバーにログインするときのパスワード
dbname	接続先データベースの名前
unix_socket	接続先 UNIX ソケットのファイル名
flags	通信プロトコルのフラグ（10 進数で指定）

  このうち dbname は必ず指定しなければならない．これらのパラメタについて詳しくは MySQL のマニュアルを参照せよ．*param* が適切でない場合は SQL.Connect 例外が発生する．

  サポートする MySQL の型と SML#の型との対応は以下の通りである．

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* および SQL.sqlite3 (*flags*, *filename*) は，SQLite3 データベースファイルへの接続情報を返す．文字列 *filename* には，データベースファイルの名前を指定する．SQLite3 は “:” で始まる名前を特別に解釈することに注意が必要である．

  flags は，以下の 4 つのフィールドからなるレコードである．

  – mode：ファイルをオープンするモードを表す．以下のいずれかの値を取る．

    * SQL.SQLite3.SQLITE_OPEN_READONLY
    * SQL.SQLite3.SQLITE_OPEN_READWRITE
    * SQL.SQLite3.SQLITE_OPEN_READWRITE_CREATE
  - threading：マルチスレッドモードを表す．以下のいずれかの値を取る．
    * SQL.SQLite3.SQLITE_OPEN_NOMUTEX
    * SQL.SQLite3.SQLITE_OPEN_FULLMUTEX
  - cache：キャッシュモードを表す．以下のいずれかの値を取る．
    * SQL.SQLite3.SQLITE_OPEN_SHAREDCACHE
    * SQL.SQLite3.SQLITE_OPEN_PRIVATECACHE
  - uri：ファイル名の解釈の仕方を表す．以下の値を取る．
    * SQL.SQLite3.SQLITE_OPEN_URI

  これらのフラグについての詳細は SQLite3 C/C++ API のマニュアルを参照せよ．これらの定数は全て SQL.SQLite3 ストラクチャに定義されている．また，SQL.SQLite3 ストラクチャは *flags* が未指定の場合に用いられるデフォルトのフラグ SQL.SQLite3.flags を持つ．フィールドアップデート式と SQL.SQLite3.flags を用いることで，デフォルトの設定を一部変更したフラグを作ることができる．

  SQLite3 の type affinity と SML#の型が以下の通りに対応付けられている．

Type affinity	SML#
INT	int
REAL	real
STRING	string
NUMERIC	numeric
BLOB	(未対応)

  CREATE TABLE 時に各カラムに指定された型と type affinity との対応は，SQLite3 のマニュアルに記載されている通りである．

- val odbc : string -> backen

  SQL.odbc *param* は，ODBC サーバーへの接続情報を返す．文字列 *param* には，DSN 名，ユーザー名，パスワードをこの順にスペースで区切って並べる．*param* が適切でない場合は SQL.Connect 例外が発生する．

  サポートする ODBC の型と SML#の型との対応は以下の通りである．

ODBC	SML#
CHAR	string
INTEGER, SMALLINT	int
FLOAT	real32
DOUBLE	real
VARCHAR, LONGVARCHAR, NVARCHAR	string

- val connect : 'a server -> 'a conn

  SQL.connect *server* は，接続情報 *server* が表す接続先のサーバーとの接続を確立し，実際のデータベーススキーマが *server* が表すスキーマを包含することを検査し，エラーが無ければ，サーバーへの接続ハンドルを返す．サーバーとの接続にエラーがあれば SQL.Exec 例外が発生する．スキーマの検査にエラーがあれば SQL.Link 例外が発生する．

接続情報 *server* の型は，この接続を通じて扱うテーブルおよびビューの型を表す．SQL.connect は，*server* の型にある各テーブル名について，同名のテーブルまたはビューが存在するかどうかを，データベースのシステムカタログを見てチェックする．存在するならば，さらにそのテーブルまたはビューが，*server* の型にあるカラムを全て持つことを検査する．チェックの際，名前の大文字小文字は区別しない．データベースは，*server* の型に無いテーブルやビューを含んでもよい．一方，各テーブルは，*server* の型に書かれているカラムを正確に含まなければならない．

初めてデータベースサーバーと接続しようとしたとき，接続先データベースの種類に応じて適切な接続ライブラリが動的にリンクされる．デフォルトのライブラリ名が適切でない場合は，環境変数を通じてライブラリ名を指定することができる．接続先サーバー，デフォルトのライブラリ名，および環境変数の名前は以下の通りである．

データベース	ライブラリ名	環境変数
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`

  以下の点を除いて SQL.connect と同じである．引数の型で指示されたテーブルが接続先データベースに存在しない場合，SQL.connectAndCreate は CREATE TABLE コマンドを発行しそのテーブルを作成する．サポートされない型を含むテーブルを作成しようとしたとき SQL.Link 例外が発生する．CREATE TABLE コマンドの実行に失敗したとき SQL.Exec 例外が発生する．

- `val closeConn : 'a conn -> unit`

  SQL.closeConn *conn* は，SQL.connect 関数が確立したデータベースサーバーへの接続を閉じる．SQL.connect で確立した接続は，SQL.closeConn で必ず閉じられなければならない．

22.8.2 SQL クエリの実行と結果の取得

- `exception Exec of string`

  SQL クエリの実行時にデータベースサーバーがエラーを報告したことを表す例外．string はエラーメッセージである．

- `val fetch : 'a cursor -> 'a option`

  SQL.fetch *cursor* は，カーソル *cursor* が指す行を一行読み込み，*cursor* を一行進める．もしカーソルがテーブルの終端に到達していた場合は NONE を返す．もしカーソルが閉じられているならば SQL.Exec 例外が発生する．

- `val fetchAll : 'a cursor -> 'a list`

  SQL.fetchAll *cursor* は，カーソル *cursor* が指す行から最後の行までを読み込み，カーソルを閉じる．もしカーソルが閉じられているならば SQL.Exec 例外が発生する．

- `closeCursor : 'a cursor -> unit`

  closeCursor *cursor* は，SQL 実行関数が返したカーソルを閉じる．全てのカーソルはこの関数か fetchAll 関数で閉じられなければならない．

22.8.3 SQL クエリの操作

- `val queryCommand : ('a list, 'b) query -> ('a cursor, 'b) command`

  `SQL.queryCommand` *query* は，SELECT クエリ *query* を SQL コマンドに変換する．この関数は，SQL 実行関数式 `_sql` ⟨*pat*⟩ `=> select...(`⟨*exp*⟩`)` がクエリ ⟨*exp*⟩ に対して行うコマンド化（22.7 節参照）と同じことを SQL 実行関数を生成せずに行う．

- `val toy : (('a, 'c) db -> ('b, 'c) query) -> 'a -> 'b`

  `SQL.toy` *query data* は，SML#のデータ構造 *data* をデータベースに見立てて SELECT クエリ *query* を評価する．クエリの評価は SML#の中で行われ，サーバーとのやり取りは一切行われない．この関数は，SML#コンパイラが SQL の型付けのために生成したトイプログラムをそのまま実行する．クエリの実行効率は考慮されていないため，この関数は実用に適さないほど遅い可能性がある．

- `val commandToString : (('a,'c) db -> ('b,'c) command) -> string`

  `SQL.commandToString` *command* は，関数 *command* の本体を評価して得られた SQL コマンドを文字列化して返す．この関数が返す文字列は，SQL 実行関数が SQL コマンド実行時にサーバーに送信する文字列と同じものである．

- `val queryToString : (('a,'c) db -> ('b,'c) query) -> string`

  `SQL.queryToString` *query* は，関数 *query* の本体を評価して得られた SELECT クエリを文字列化する．この関数が返す文字列は，SQL 実行関数が SELECT クエリ実行時にサーバーに送信する文字列と同じものである．

- `val expToString : ('a,'c) exp -> string`

  `SQL.expToString` *exp* は，SQL 評価式 *exp* を文字列化する．

22.9 SQL ライブラリ：SQL.Op ストラクチャ

`SQL.Op` ストラクチャは，SQL 評価式の中で用いる SQL の二項演算子や集約関数，およびその他のユーティリティーを提供する．SQL 評価式構文の中では，いくつかの識別子が以下の通り infix 宣言される．

```
infix 7 %
infix 5 like ||
nonfix mod
```

SML#で評価される式でない SQL 評価式構文内では，このストラクチャで定義されている関数や演算子をストラクチャ名を前置することなく使用することができる．

`SQL.Op` ストラクチャで定義されている関数の多くは，複数の SQL 基本型（22.1 節参照）に対してオーバーロードされている．本節では，説明の便宜上，オーバーロードされている型とその動く範囲を，以下の型変数の名前で示す．

- `'sql` は全ての SQL 基本型およびその `option` 型を動く．
- `'sqlopt` は全ての SQL 基本型の `option` 型を動く．
- `'num` は SQL 基本型のうち数値型およびその `option` 型を動く．
- `'str` は `string` 型および `string option` 型のどちらかを動く．
- これら以外の型変数は，通常通り，任意の型を動く．

`SQL.Op` ストラクチャのシグネチャは以下の通りである．

```
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
```

以下，これらの定義を役割ごとに副節に分けて説明する．

22.9.1 型を合わせるための何もしない関数

SML#がサポートしない暗黙のキャストやオーバーロードに対応するため，型の辻褄を合わせるための以下の関数が提供されている．これらの使い方は 22.4.5 節を参照せよ．

- `val Some : 'a -> 'a option`
- `val Part : 'a option list -> 'a list`
- `val Num : 'num -> numeric option`

22.9.2 SQL 演算子および関数

SQL.Op ストラクチャは以下の二項演算子を提供する．これらの演算子は，引数に取る SQL 評価式に同名の演算子を加えた SQL 評価式を返す．演算子式はクエリがサーバーに送信されるまで評価されない．

- 比較演算子：5 種類の比較演算子<, >, <=, >=, =, <>が全ての SQL 基本型に対してオーバーロードされている．これらの比較演算子の型は，

  ```
  'sql * 'sql -> bool3
  ```

  である．

- 算術演算子：4 種類の二項演算子+, -, *, /, % および 2 つの単項演算子~, abs が全ての SQL の数値型に対してオーバーロードされている．SQL 評価式の中では，識別子%は infix と宣言される．これらの型は，

  ```
  'num * 'num -> 'num
  ```

  または

  ```
  'num -> 'num
  ```

  である．

- 剰余演算：剰余演算子 mod は，標準 SQL に合わせ，二項演算子ではなく関数としても提供されている．接続先のデータベースによっては，mod と%のどちらか片方しかサポートしていないことに注意すること．SQL 評価式の中では，識別子 mod は nonfix と宣言される．mod の型は，

  ```
  'num * 'num -> 'num
  ```

  である．

- 文字列演算子：パターンマッチ演算子 like と結合演算子||が使用できる．どちらの識別子も SQL 評価式の中では infix 宣言されている．これらの型は以下の通りである．

  ```
  val like : 'str * 'str -> bool3
  val || : 'str * 'str -> 'str
  ```

- NULLIF：以下の型の nullif 関数が提供されている．

  ```
  val nullif : 'sqlopt * 'sqlopt -> 'sqlopt
  ```

  引数は 2 つとも option 型でなければならない．適宜 Some で型を合わせること．

- COALESCE: option 型の取り扱いの違いで，coalesce と coalesce' の 2 つの関数が提供されている．

  ```
  val coalesce : 'b option * 'b -> 'b
  val coalesce' : 'b option * 'b option -> 'b option
  ```

  どちらの関数を用いても，サーバーに送信される関数名は COALESCE である．coalesce の型は，NULL を NULL でない値で置き換える使い方に合わせて決めている．標準 SQL とは異なり，3 つ以上の引数はサポートしていない．3 つ以上の値を取り扱うときは coalesce' 関数をネストすること．

22.9.3 SQL 集約関数

5 つの標準 SQL の集約関数 count，avg，sum，min，max が利用可能である．このうち，sum，min，max 関数は，option 型の取り扱いの違いにより，2 種類の関数（例えば sum と sum'）が提供されている．どちらの関数を用いても，サーバーに送信される関数名は同じである．

これらの関数の型は以下の通りである．

```
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 ライブラリ：SQL.Numeric ストラクチャ

SQL.Numeric ストラクチャは，標準 SQL で最大の精度を持つ数値型 NUMERIC をエミュレートする．

SQL.Numeric のシグネチャは以下の通りである．

```
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
```

これらの関数の意味は Basis Library の `Int` および `IntInf` ストラクチャに定義されている同名の関数と同じである．`toString` 関数は小数点以下に最大で 16,383 文字を出力するかもしれないことに注意せよ．

`SQL.Numeric` ストラクチャはあくまでデータベースとのデータのやり取りのために用意されたものであり，一般的な無限精度の 10 進数ライブラリを提供することを意図していない．数値演算の性能を追求して実装されていないため，これらの関数は実用に適さないほど遅い可能性がある．

なお，`SQL.numeric` 型の別名として `SQL.decimal` 型が，`SQL.Numeric` の別名として `SQL.Decimal` ストラクチャが，それぞれ提供されている．

22.11　標準 SQL 文法との差異（参考）

SML#の SQL 演算子式の構文は，可能な限り標準 SQL の構文に似せている．そうすることによって，SML#の SQL 機能のために新しい埋め込み言語を覚えなくてもよいようにしている．しかし，埋め込み言語の性質上，ホスト言語の文法の影響を受けて，標準 SQL の文法を実現できない場合がある．以下に，SML#の SQL 構文が標準 SQL と異なっている箇所をまとめる．以下の点に気をつければ，標準 SQL ほぼそのままの書き方で SML#の SQL 機能を使いこなせるはずである．

- 定数リテラルの書き方（特に文字列）は SML#に準じる．
- 標準 SQL ではキーワードの大文字小文字は無視される一方，SML#では全ての SQL キーワードは小文字でなければならない．
- テーブル名やカラム名の大文字小文字は SML#の型システム上では区別される．ただし，`SQL.connect` がスキーマの一致を検査するときのみ，名前の大文字小文字は区別されない．
- 標準 SQL では関数適用構文で引数を囲む必要であるが，SML#では SML#の関数適用式と同様に括弧を書かなくてよい．
- 負符号演算子には SML#式と同様に~を用いる．
- 二項演算子の結合順位は SML#の infix 宣言によって決まる．
- and 論理演算子は括弧に囲まれた式の中でしか使えない．
- カラムやテーブルの参照には#を前置する．
- カラムの参照には必ずテーブル名を明示しなければならない．
- ORDER BY 句から SELECT 句のカラム名を参照する場合は，カラム名に#. を前置する．
- FROM 句や INSERT，UPDATE，DELETE コマンドのテーブル参照には，#と，テーブルが属するデータベースを指す SML#の識別子（一般に `_sql` や `fn` 構文で束縛される）を前置する．

第23章　核言語の宣言とインターフェイス

宣言 n の種類毎に，宣言 ⟨*decl*⟩ の定義と，それに対応するインターフェイス仕様 ⟨*interfaceSpec*⟩ の定義を与え，それら宣言に対してコンパイラが計算する型と値を説明する．

23.1　val 宣言：⟨*valDecl*⟩

val 宣言の構文は，以下の文法で定義される．

⟨*valDecl*⟩ ::= `val` ⟨*tyvarSeq*⟩ ⟨*valBind*⟩
⟨*valBind*⟩ ::= ⟨*valBind1*⟩
| ⟨*valBind1*⟩ `and` ⟨*valBind*⟩
⟨*valBind1*⟩ ::= ⟨*pat*⟩ `=` ⟨*exp*⟩

この宣言では，`and` で接続された複数の宣言に含まれる変数が同時に束縛される．束縛された変数の有効範囲は，この宣言と同一のスコープを持つこの宣言に続く部分である．各 ⟨*pat*⟩ に現れる変数はすべて異なっていなければならない．

⟨*valDecl*⟩ の ⟨*tyvarSeq*⟩ は，⟨*valBind*⟩ に現れる型変数のスコープの指定である．これら型変数は，各 ⟨*valBind1*⟩ のトップレベルで型抽象される．

23.1.1　val 宣言インタフェイス：⟨*valSpec*⟩

val 宣言のインターフェイスは，束縛される変数それぞれについて，とその型を以下の形で宣言する．

⟨*valSpec*⟩ ::= `val` ⟨*id*⟩ `:` ⟨*ty*⟩

例えば，val 宣言

```
val (x,y) = (1,2)
```

に対する val 宣言インターフェイスは以下の２つのインターフェイス宣言である．

```
val x : int
val y : int
```

23.1.2　val 宣言の評価

値束縛宣言

`val` ⟨*pat*⟩$_1$ `=` ⟨*exp*⟩$_1$ `and` ⟨*pat*⟩$_2$ `=` ⟨*exp*⟩$_2$ $\cdots$

の評価は，構造パターンの評価と式の評価の２段階で行われる．

構造パターン評価

構造パターンを含む値束縛は，まず，パターン集合に含まれる変数集合 $\{x_1, \ldots, x_m\}$ に対する値束縛に変換される．

パターン集合に定数やコンストラクタパターンが一つも含まれない場合，この変換は以下の手順で行われる．

1. パターン $\langle pat_i \rangle$ と式 $\langle exp_i \rangle$ の組が，再帰的に，部分パターンと部分式の組みの集合に分解される．
2. 各パターン $\langle pat_i \rangle$ が，その構造に従い，部分パターンの組に分解される．この時，$\langle pat_i \rangle$ が型制約を持てば，分解された部分パターンにも型制約が付加される．
3. 式の分解は，上記ステップで分解された部分パターンに対して，対応する部分を取り出す操作を，式 $\langle exp_i \rangle$ に対して静的に実行することによって行う．すなわち，式が構造式の場合は，部分式が取り出され，式が構造式でない場合は，式を変数に束縛し，構造から部分構造を取り出す関数を変数に適用する式が生成される．
4. パターンが id `as` pat の形の階層パターンを含む場合，上記の操作に加えて，id に対して構造全体の値が静的に生成される．
5. 以上生成された変数と式の組の集合が，

   `val` x_1 $(: \tau_1)?$ `=` exp_1
   $\cdots$
   `and` x_m $(: \tau_m)?$ `=` exp_m

   の形の変数束縛宣言として生成される．

以下の 23.1.3 で示すように，以上のようにパターンによる値束縛宣言を，可能な限り変数の束縛に変換することにより，`Bind` 例外を起こさない値束縛において，SML#が推論する式のランク 1 多相型を変数の型として引き継ぐことが可能となる．

パターン集合に定数やコンストラクタパターンが含まれる場合，構造を含む値束縛は，実行時に例外を発生することがあり，静的な評価をすることができない．この場合は，与えられた値束縛が以下の形の変数束縛に変換される．

```
val X = case (⟨exp1⟩, ..., ⟨expn⟩) of
          (⟨pat1⟩, ..., ⟨patn⟩) => (x1,...,xm)
        | _ => raise Bind
val x1 = #1 X
  ...
and xm = #m X
```

以上の変換により，値束縛宣言に含まれる変数集合の式への同時束縛宣言が作られる．

23.1.3　val 宣言とインタフェイスの例

以下は，ランク 1 多相性を含む型束縛の例である．

```
# 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]]
```

このように，変数には式の持ちうるランク 1 多相が与えられる．この評価方式は，上例のように式の一部に副作用を持つ場合でも，安全である．以下の例のように，パターンがコンストラクタや定数を含む場合は，この値束縛構文自体が副作用を含む可能性があるため，変数への多相型は与えられない．

```
# val (x,y, 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 x = () : unit
val y = fn : fn : ?X7 -> ?X6 -> ?X7 * ?X6
```

以下は，ソースファイルとインターフェイスファイルの簡単な例である．

Version.sml file:

```
val (version, releaseDate) = ("3.7.1", "2021-03-15")
```

Version.smi file:

```
val version : string
val releaseDate : string
```

23.2 関数宣言：⟨valRecDecl⟩, ⟨funDecl⟩

関数宣言の構文には以下の ⟨valRecDecl⟩ 及び ⟨funDecl⟩ がある．

⟨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$_m$⟩

この宣言で，`and` で接続された複数の関数が相互再帰的に定義される．定義される関数名の有効範囲は，この宣言全体と，この宣言に続く部分である．

⟨valRecDecl⟩ における ⟨valBind⟩ の val 宣言は，

⟨vid⟩ `=` `fn` 式

の形に制限される．

同一の ⟨funBind1⟩ に現れる識別子（関数名）⟨vid⟩ はすべて同一でなければならず，異なる ⟨funBind1⟩ に現れる関数名 ⟨vid⟩ は互いに異ならなければならない．また，同一の ⟨funBind1⟩ に現れる ⟨pat⟩ に含まれる変数はすべて異なっていなければならない．

さらにこの宣言の評価においては，関数名 ⟨vid⟩ が，定義される関数と同一の値（静的型および動的な値）をもつ変数と定義される．

⟨tyvarSeq⟩ は，⟨valBind⟩ および ⟨funDecl⟩ に現れる型変数のスコープの指定である．これら型変数は，各 ⟨funBind1⟩ および ⟨valBind⟩ の中の各 ⟨valBind1⟩ のトップレベルで型抽象される．

23.2.1 関数宣言インターフェイス

関数宣言のインターフェイスは，val 宣言のインターフェイスと同一である．関数宣言で定義される関数名と型を val 宣言のインターフェイスと同一の構文で記述する．以下は，関数宣言に対するソースファイルとインターフェイスの例である．

`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 宣言：⟨*datatypeDecl*⟩

datatype 宣言は，新しい型構成子を定義する．その構文は以下の通りである．

⟨*datatypeDecl*⟩ ::= `datatype` ⟨*datbind*⟩ (`withtype` ⟨*tybind*⟩)?
⟨*datbind*⟩ ::= (⟨*tyvarSeq*⟩)? ⟨*tycon*⟩ `=` ⟨*conbind*⟩ (`and` ⟨*datbind*⟩)?
⟨*conbind*⟩ ::= (`op`)? ⟨*vid*⟩ (`of` ⟨*ty*⟩)? (`|` ⟨*conbind*⟩)?
⟨*tybind*⟩ ::= ⟨*tyvarSeq*⟩ ⟨*tycon*⟩ `=` ⟨*ty*⟩ (`and` ⟨*tybind*⟩)?

型構成子名 ⟨*tycon*⟩ はすべてて異なっていなければならず，同一の ⟨*datbind*⟩ の中は，データ構成子名 ⟨*vid*⟩ はすべて異なっていなければならない．この宣言により，型変数 (⟨*tyvarSeq*⟩)? をパラメタとする相互再帰的な多相型型構成子 ⟨*tycon*⟩ と，データ構成子 ⟨*vid*⟩ が定義される．⟨*tycon*⟩ のスコープは，⟨*datatypeDecl*⟩ 全体と，この宣言に続く宣言である．`withtype` は，⟨*datatypeDecl*⟩ をスコープとする型の別名定義であり，評価前に展開される．

23.3.1 datatype 宣言インタフェース

datatype宣言に対するインタフェイスには，datatype仕様と抽象型仕様がある．datatype仕様は，datatype宣言と同一である．抽象型仕様の構文は以下の通りである．

⟨*opequeTypeSpec*⟩ ::= `type` (⟨*tyvarSeq*⟩)? ⟨*tycon*⟩ (`=` ⟨*runtimeTypeSpec*⟩)
⟨*runtimeTypeSpec*⟩ := `unit` | `contag` | `boxed`

datatype 宣言に対して datatype 宣言インターフェイスを記述すると，`_require` を通じて参照するコンパイル単位に対して，datatype 宣言された型構成子とデータ構成子が定義される．datatype 宣言に対して抽象型インタフェイスを記述すると，同一名前の抽象型構成子が定義される．抽象型インタフェイスのオペランドは，その型の実行時表現をコンパイラに通知するための注釈であり，datatype 宣言に応じて `unit`, `contag`，または `boxed` のいずれかを記述する．`unit` および `contag` は，datatype の実行時表現が構成子タグのみであるを意味する．datatype 宣言に引数を持つデータ構成子が無い場合に記述する．構成子が 1 つだけのとき `unit` を，そうでないとき `contag` を書く．`boxed` は，datatype の実行時表現がポインタであることを意味する．datatype 宣言に引数を持つデータ構成子がある場合に記述する．

23.3.2 datatype 宣言とインタフェイスの例

以下は，datatype 宣言を含むソースファイルとインターフェイスの例である．

`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 宣言：⟨*typDecl*⟩

type 宣言は，以下の文法で型に名をつける宣言である．

⟨*typeDecl*⟩ := `type` ⟨*typbind*⟩
⟨*tybind*⟩ ::= ⟨*tyvarSeq*⟩ ⟨*tycon*⟩ `=` ⟨*ty*⟩ (`and` ⟨*tybind*⟩)?

23.4.1 type 仕様：⟨*typSpec*⟩

type 宣言のインタフェイスは，type 仕様と抽象型仕様がある．type 仕様の構文は type 宣言と同一である．抽象型仕様の構文は以下の通りである．

⟨*opequeTypeSpec*⟩ := `type` (⟨*tyvarSeq*⟩)? ⟨*tycon*⟩ `(` `=` ⟨*runtimeTypeSpec*⟩ `)`
⟨*runtimeTypeSpec*⟩ := ⟨*tycon*⟩ | `{}` | `*` | `->`

型がレコード型，組型，関数型で実装されているならば，⟨*runtimeTypeSpec*⟩ にはそれぞれ`{}`，`*`，`->`を指定する．そうでなければ，型を実装する型コンストラクタを ⟨*runtimeTypeSpec*⟩ に指定する．実行時表現さえ合っているならば，⟨*runtimeTypeSpec*⟩ は実装型と厳密に一致しなくてもよい．実行時表現について詳しくは 29 章を参照せよ．

23.4.2 型宣言とインタフェイスの例

以下は，型宣言を含むソースファイルとインターフェイスの例である．

`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　例外宣言：⟨*exnDecl*⟩

例外宣言の構文は以下のとおりである．

⟨*exnDecl*⟩　::=　`exception` ⟨*exbind*⟩
⟨*exbind*⟩　::=　(`op`)? ⟨*vid*⟩ (`of` ⟨*ty*⟩)? (`and` ⟨*exbind*⟩)?

この宣言により型 ⟨*ty*⟩ を引数とする例外構成子 ⟨*vid*⟩ が定義される．

23.5.1　例外仕様：⟨*exnSpec*⟩

例外のインタフェイスである例外仕様の構文は，例外宣言と同一である．

23.5.2　例外宣言とインタフェイスの例

以下は，例外宣言を含むソースファイルとインターフェイスの例である．

`Data.sml` file:

```
exception Fail of string
```

`Data.smi` file:

```
exception Fail of string
```

第24章　モジュール言語の宣言とインタフェイス

宣言の種類毎に，宣言 ⟨*topdecl*⟩ の定義と，それに対応するインターフェイス仕様 ⟨*interfaceSpec*⟩ の定義を与え，それら宣言に対してコンパイラが計算する型と値を説明する．

24.1　ストラクチャ宣言：⟨*strDecl*⟩

ストラクチャ宣言は，ストラクチャ名 ⟨*strid*⟩ を，ストラクチャ式 ⟨*strexp*⟩ 表すストラクチャに束縛する形の宣言である．

⟨*strDecl*⟩ ::= `structure` ⟨*strbind*⟩
⟨*strbind*⟩ ::= ⟨*strid*⟩ `=` ⟨*strexp*⟩ (`and` ⟨*strbind*⟩)?
| ⟨*strid*⟩ `:` ⟨*sigexp*⟩ `=` ⟨*strexp*⟩ (`and` ⟨*strbind*⟩)?
| ⟨*strid*⟩ `:>` ⟨*sigexp*⟩ `=` ⟨*strexp*⟩ (`and` ⟨*strbind*⟩)?

`and` で繋がれた複数のストラクチャ名は，同時に束縛される．それら名前のスコープは，この宣言に続く部分である．従って，以下の宣言では，`y` は `2` ではなく `1` に束縛される．

```
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;
```

シグネチャ制約付きのストラクチャ名束縛は，以下のシグネチャ制約付きのストラクチャ式への束縛へと変換される．

変換前	変換後
⟨*strid*⟩ `:` ⟨*sigexp*⟩ `=` ⟨*strexp*⟩	⟨*strid*⟩ `=` ⟨*strexp*⟩ `:` ⟨*sigexp*⟩
⟨*strid*⟩ `:>` ⟨*sigexp*⟩ `=` ⟨*strexp*⟩	⟨*strid*⟩ `=` ⟨*strexp*⟩ `:>` ⟨*sigexp*⟩

ストラクチャ宣言の評価は以下のように行われる．

1. 次節の定義に従い，ストラクチャ式を評価して静的型環境 Γ と実行時環境 E を求める．
2. 静的環境 Γ と動的 E で束縛されたロング名をストラクチャ名 S でプレフィックスした環境 Γ' と E' を求め現在の環境に追加する．

例えばストラクチャ宣言 `structure` ⟨*strid*⟩ `=` ⟨*strexp*⟩ において，ストラクチャ式 ⟨*strexp*⟩ が生成する型環境と実行時環境が $\{longId_1 : \tau_1, ..., longid_n : \tau_n\}$ および $\{longId_1 : v_1, ..., longid_n : v_n\}$ であれば，このストラクチャ宣言の効果は，現在の静的型環境および実行時環境に，$\{S.longId_1 : \tau_1, ..., S.longid_n : \tau_n\}$ および $\{S.longId_1 : v_1, ..., S.longid_n : v_n\}$ のロング名束縛が追加される．

24.2　ストラクチャ式とその評価：⟨*strexp*⟩

ストラクチャ式の構文は以下の通りである．

$$
\begin{array}{lcll}
\langle strexp \rangle & ::= & \texttt{struct}\ \langle strdec \rangle\ \texttt{end} & \\
& | & \langle longStrid \rangle & \\
& | & \langle strexp \rangle\ \texttt{:}\ \langle sigexp \rangle & \\
& | & \langle strexp \rangle\ \texttt{:>}\ \langle sigexp \rangle & \\
& | & \langle funid \rangle\ \texttt{(}\ \langle strid \rangle\ \texttt{:}\ \langle sigexp \rangle\ \texttt{)} & \\
& | & \langle funid \rangle\ \texttt{(}\ \langle strdec \rangle\ \texttt{)} & \\
\langle strdec \rangle & ::= & \langle decl \rangle & \text{核言語の宣言} \\
& | & \texttt{structure}\ \langle strbind \rangle & \\
& | & \texttt{local}\ \langle strdec \rangle\ \texttt{in}\ \langle strdec \rangle\ \texttt{end} & \\
& | & \langle strdec \rangle\ (\texttt{;})?\ \langle strdec \rangle &
\end{array}
$$

各ストラクチャ式の評価は以下のように行われる．

- 基本ストラクチャ式 : `struct` ⟨*strdec*⟩ `end`.

  基本ストラクチャ式の評価は，宣言 ⟨*strdec*⟩ を順次評価することよって行われる．その結果は，核言語の宣言に対しては，第 23 章での定義にしたがい，変数や型構成子等の名前の束縛が生成される．ストラクチャ宣言に対しては，上記の評価規則に従い，ストラクチャ名からなるロング名の束縛が生成される．

- ロングストラクチャ名 : ⟨*longStrId*⟩ .

  現在の環境において，⟨*longStrId*⟩ が束縛されている型環境と実行時環境が返される．現在の束縛の集合の中で ⟨*longStrId*⟩ プリフィックをもつロング名の束縛から，⟨*longStrId*⟩ プリフィックを取り除いてえられる束縛の集合と同一である．

- シグネチャ制約付きストラクチャ式. (⟨*strexp*⟩ `:` ⟨*sigexp*⟩ , ⟨*strexp*⟩ `:>` ⟨*sigexp*⟩)

  シグネチャ制約付きストラクチャ式の評価は以下のように行われる．

  1. ストラクチャ式を評価し，静的な型環境 Γ を求める．
  2. 次項 (24.3) の定義に従いシグネチャ式を評価し，ロング名の静的な値の制約の集合 Σ を求める．
  3. Σ に含まれる各ロング名の静的制約に対しして，Γ にロング名束縛が存在し，かつそのその静的値が制約を満たすことをチェックする．
  4. Γ を Σ で指定されたロング名に制約しや型環境 Γ′ を生成する．
  5. もし ⟨*strexp*⟩ `:>` ⟨*sigexp*⟩ の形の参照不透明なシグネチャ制約であれば，さらに，シグネチャの型指定 ⟨*typdesc*⟩ に対応する Γ′ の型を，抽象型で置き換えて得られる型環境 Γ″ を生成する．
  6. 実行時環境の生成は，動的な値の生成はシグネチャ制約がないストラクチャ式と同一であるが，生成される動的束縛は Γ″ に含まれる変数とコンストラクタに限定される．

24.3　シグネチャ式 : ⟨*sigexp*⟩

シグネチャ式の文法は以下の通りである．

$$
\begin{array}{lcl}
\langle sigexp \rangle & ::= & \texttt{sig}\ \langle spec \rangle\ \texttt{end} \\
& | & \langle sigid \rangle \\
& | & \langle sigexp \rangle\ \texttt{where type}\ \langle tyvarSeq \rangle\ \langle longTycon \rangle\ \texttt{=}\ \langle ty \rangle
\end{array}
$$

- 基本シグネチャ式 (`sig` ⟨*spec*⟩ `end`)

  宣言仕様 ⟨*spec*⟩ の列である．

- シグネチャ名 (⟨*sigid*⟩)

  トップレベルのシグネチャ宣言でシグネチャ式に束縛された名前である．

- 型定義付きのシグネチャ名

  シグネチャ名 ⟨*sigid*⟩ に束縛されたシグネチャ式の中の ⟨*tyvarSeq*⟩ ⟨*longTycon*⟩ を ⟨*ty*⟩ で置き換えて得られるシグネチャ式を表す．

宣言仕様 ⟨*spec*⟩ の構文は以下の通りである．

⟨*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*⟩)?

それぞれの要素の意味は以下の通りである．

- `val` ⟨*valdesc*⟩．val 宣言で束縛される変数の型を指定する．

- `type` ⟨*typdesc*⟩．指定されたロング名をもつ type 宣言または datatype 宣言が存在することの指定である．

  参照透明なシグネチャ指定の場合，この仕様は，ストラクチャの type 宣言で定義される型関数をそのまま有効にする．datatype 宣言の場合，ストラクチャの型構成子が有効になるが，データ構成子は束縛されない．

  参照不透明なシグネチャ指定の場合，この仕様は，宣言で定義されるの型関数の内部構造および datatype で生成された型構成子の同一性を隠蔽し，抽象型構成子として使用可能とする．

- `eqtype` ⟨*typdesc*⟩

  `type` ⟨*typdesc*⟩ と同一であるが，ストラクチャで宣言される型が同値演算が可能な型に限定される．

- `datatype` ⟨*datdesc*⟩

  同一の datatype 宣言が存在することの指定である．datatype 宣言が有効になる．

- `datatype` ⟨*tycon*⟩ `=` `datatype` ⟨*longTycon*⟩

  型構成子名 ⟨*tycon*⟩ に束縛された型構成子が，⟨*longTycon*⟩ に束縛されたものと同一であることの指定．

- `exception` ⟨*exdesc*⟩

  exception 宣言が存在することの指定である．exception 宣言が有効になる．

- `structure` ⟨*strdesc*⟩

  ストラクチャが指定されたシグネチャ式を持つストラクチャを含むことの指定である．

- `include`

  指定されたシグネチャ名に束縛されたシグネチャ式の内容（⟨*spec*⟩ の列）がこの位置に展開される．

- `sharing type`

  指定されたロング型構成子名がすべて同じ型構成子に束縛されていることの指定である．

24.4　モジュール言語のインタフェイス

モジュール言語のインタフェイスは，第 16.2 節で定義した通り，ストラクチャのプロバイド（⟨*provideStr*⟩）とファンクタのプロバイド（⟨*provideFun*⟩）の２つである．それらの構文は以下の通りである．

⟨*provideStr*⟩	::=	`structure` ⟨*strid*⟩ `=` `struct` ⟨*provideStrdecl*⟩ `end`
⟨*provideStrdecl*⟩	::=	⟨*provideVal*⟩
	|	⟨*provideType*⟩
	|	⟨*provideDatatype*⟩
	|	⟨*provideException*⟩
	|	⟨*provideStr*⟩
⟨*provideFun*⟩	::=	`functor` ⟨*provideFunBind*⟩
⟨*provideFunBind*⟩	::=	⟨*funid*⟩ `(` ⟨*strdesc*⟩ `)` `=` ⟨*provideStrExp*⟩

以下はストラクチャ宣言を含むソースファイルとインターフェイスの例である．

`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
```

第25章　SML#のライブラリ概要

関数型言語であるSML#では，種々のデータ構造とその操作は，型 (datatype) の定義とそれを操作する関数集合を含んだストラクチャとして定義される．さらに，SML#の分割コンパイル機能によって，関連するストラクチャがファイルにまとめられ，それに対するインターフェイスファイル（smi ファイル）が定義されている．本章では，インターフェイスファイルにまとめられたストラクチャ群をライブラリと呼ぶ．関連するライブラリは，インタフェイス言語の`_include`文によってまとめられ，階層化されている．

SML#が提供するライブラリは以下に大別される．

- Standard ML 標準ライブラリ

  Standard ML ライブラリ仕様 [2] で定義されたライブラリである．組み込みデータの操作や標準の IO などが提供されている．SML#では，Standard ML Basis Library の必須ライブラリのすべてとオプションライブラリの一部が提供されている．

- SML#ライブラリ

  C との直接連携，SQL 統合，JSON サポートなどの SML#に組み込まれた機能を使う上で有用な以下のサポートライブラリである．

  - FFI ライブラリ
  - SQL ライブラリ
  - Thread ライブラリ
  - Reify ライブラリ

- ユーテリティ群

  サードパーティ製を含む種々の汎用で有用な関数が提供される．以下のものを含む．

  - SML New Jersey ライブラリ

- ツールのサポートライブラリ

  SML#が提供する構文解析ツール，清書プログラム生成ツール，テストツールなどを，SML#から使用するためのライブラリである．以下のものを含む．

  - smlyacc と smllex のサポートライブラリ
  - 清書ツール SMLFormat サポートライブラリ
  - ユニットテストツール SMLUnit サポートライブラリ

本ドキュメントでは，ライブラリの仕様を以下の形式で定義する．

1. Standard ML 標準ライブラリなど，Standard ML のシグネチャが公式に定義されいるライブラリについては，シグネチャの定義を示し，さらに，各ストラクチャについて，シグネチャでは表現されない型インスタンスの情報を追記する．
2. 主に SML#独自のライブラリなど，標準のシグネチャが定義されていないライブラリは，そのインタフェイス情報を示す．インタフェイスは，シグネチャ情報に加えて，分割コンパイルでリンクして使用する上で必要な情報を含んでいる．

第26章　Standard ML標準ライブラリ

SML#は，Standard ML Basis Library [2] で定義されている必須のストラクチャを標準で提供している．これらすべてのストラクチャには，公式のシグネチャが定義されている．そこで本節では，これら公式シグネチャを定義し，各シグネチャ定義に続き，そのシグネチャを実装するストラクチャを，必要な型の定義とともに示す．本節で説明する各関数の操作仕様の詳細な情報は，Standard ML Basis Library [2] を参照せよ．

SML#が提供する（トップレベルの）シグネチャとストラクチャは以下の通りである．

シグネチャ名	シグネチャを実装するストラクチャ	節
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)

これらすべてのライブラリのインターフェースファイルは，階層的にまとめられ，ライブラリインタフェイスファイル basis.smi にまとめられている．従って，以下の_require 宣言を，ソースプログラムのインタフェイスファイルに書けば，そのソースプログラムの中で，シ Standard ML 標準ライブラリのすべてのストラクチャとシグネチャが使用可能となる．

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

26.1 ARRAY

組み込みデータ型である配列型 τ array の操作を提供する．

シグネチャ

```
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
```

シグネチャを実装するストラクチャ

- Array:ARRAY.

26.2 ARRAY_SLICE

配列の一部を表現する抽象データ型とその操作を提供.

```
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
```

シグネチャを実装するストラクチャ

- `ArraySlide : ARRAY_SLICE`

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

26.3 BIN_IO

バイト列に対する IO 処理を提供する．手続き型 IO のシグネチャ`IMPERATIVE_IO`（26.4）の拡張として定義される．

```
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
```

シグネチャを実装するストラクチャ

- 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
```

ネストしたシグネチャ

- 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

真理値データ型と基本演算を提供.

```
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
```

シグネチャを実装するストラクチャ

- Bool : BOOL

26.7 BYTE

文字とバイト表現の変換をサポート.

```
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
```

シグネチャを実装するストラクチャ

- Byte : BYTE

26.8 CHAR

文字データの基本操作を提供.

```
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
```

シグネチャを実装するストラクチャ

- Char : CHAR

26.9 COMMAND_LINE

SML#で作成した実行形式プログラムのコマンドラインデータの提供.

```
signature COMMAND_LINE =
sig
  val arguments : unit -> string list
  val name : unit -> string
end
```

シグネチャを実装するストラクチャ

- CommandLine : COMMANDLINE

26.10 DATE

日付データ型とその操作プリミティブの提供.

```
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
```

シグネチャを実装するストラクチャ

- Date :> DATE

```
type date (= boxed)
```

26.11 GENERAL

参照型と例外型のプリミティブおよびシステム共通例外名を提供.

```
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
```

シグネチャを実装するストラクチャ

- General : GENERAL

26.12 IEEE_REAL

IEEE 準拠の浮動小数点表現を提供.

```
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
```

シグネチャを実装するストラクチャ

- IEEEReal : IEEE_REAL

26.13 IO

汎用の IO 例外等を定義するストラクチャである.

```
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
```

シグネチャを実装するストラクチャ

- IO : IO

26.14 INTEGER

符号付き整数演算を提供する.

```
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.

シグネチャを実装するストラクチャ

- Int : INTEGER

  ```
  type int = int
  ```

  Int32 と Position は Int のリプリケーション.

- Int64 : INTEGER

  ```
  type int = int64
  ```

  LargeInt は Int64 のリプリケーション.

- Int8 : INTEGER

  ```
  type int = int8
  ```

26.15 INT_INF

桁数制限のない符号付き整数の操作関数.

```
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
```

シグネチャを実装するストラクチャ

- IntInf : INT_INF

  ```
  type int = intInf
  ```

26.16 LIST

組み込みのリスト型のプリミティブ関数を提供する.

```
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
```

シグネチャを実装するストラクチャ

- List : LIST

  ```
  type 'a list = 'a list
  ```

26.17 LIST_PAIR

リストのペアを扱う汎用関数を提供.

```
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
```

シグネチャを実装するストラクチャ

- ListPair : LIST_PAIR

26.18 MONO_ARRAY

単相配列型とその操作プリミティブを提供.

```
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
```

シグネチャを実装するストラクチャ

- 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

単相型配列の部分配列の操作関数を提供.

```
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
```

シグネチャを実装するストラクチャ

- 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

単相ベクトル型とその操作プリミティブを提供.

```
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
```

シグネチャを実装するストラクチャ

- CharVector : MONO_VECTOR

  ```
  type elem = char
  type vector = string
  ```

- Word8Array : MONO_ARRAY

  ```
  type elem = word8
  type vector = word8 vector
  ```

26.21 MONO_VECTOR_SLICE

単相型ベクトルの部分ベクトルの操作関数を提供.

```
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
```

シグネチャを実装するストラクチャ

- 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

オプション型とプリミティブを提供.

```
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

OS とのインターフェイスを提供.

```
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
```

ネストしたシグネチャ

- OS_FILE_SYS (26.24)
- OS_IO (26.25)
- OS_PATH (26.26)
- OS_PROCESS (26.27)

シグネチャを実装するストラクチャ

- 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

OS のファイルシステムインタフェイスを提供. OS シグネチャのサブシグネチャ.

```
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

OS の IO インタフェイスを提供．OS シグネチャのサブシグネチャ．

```
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

OS のファイルパス操作関数を提供．OS シグネチャのサブシグネチャ．

```
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

OS のプロセス・スレッド操作関数を提供．OS シグネチャのサブシグネチャ．

```
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

浮動小数点演算を提供．

```
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
```

ネストしたシグネチャ

- MATH (26.29)

シグネチャを実装するストラクチャ

- Real : REAL

  ```
  type real = real
  ```

  Real64 と LargeReal は Real のストラクチャリプリケーション.

- Real32 : REAL

  ```
  type real = real32
  ```

26.29 MATH

浮動小数点演算用の数値計算関数を提供. REAL のサブシグネチャ.

```
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

文字列型の操作関数.

```
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
```

シグネチャを実装するストラクチャ

- String : STRING

```
type char = char
type string = string
```

26.31 STRING_CVT

分文字型列処理関数を提供.

```
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
```

シグネチャを実装するストラクチャ

- `StringCvt : STRING_CVT`

  ```
  type cs (= boxed)
  ```

26.32 SUBSTRING

部分文字型とその列処理関数を提供.

```
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
```

シグネチャを実装するストラクチャ

- Substring :> SUBSTRING
  where type substring = CharVectorSlice.slice
  where type string = string
  where type char = char

26.33 TEXT

文字列ストラクチャを提供.

```
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
```

シグネチャを実装するストラクチャ

- 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
```

ネストしたシグネチャ

- TEXT_STREAM_IO (26.35)

シグネチャを実装するストラクチャ

- 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

低レベルの IO プリミティブを提供.

```
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
```

シグネチャを実装するストラクチャ

- 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

時刻データ型とプリミティブ演算を提供.

```
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
```

シグネチャを実装するストラクチャ

- Time :> TIME

```
type time (= real)
```

26.38 TIMER

タイマー型とそのプリミティブ関数を提供.

```
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
```

シグネチャを実装するストラクチャ

- Timer :> TIMERE

```
type cpu_timer (= boxed)
type real_timer (= boxed)
```

26.39 VECTOR

ベクトル（更新不可の配列）型とそのプリミティブ関数を提供する.

```
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
```

シグネチャを実装するストラクチャ

- Vector : VECTOR

26.40 VECTOR_SLICE

部分ベクトル型とそのプリミティブ演算の提供.

```
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
```

シグネチャを実装するストラクチャ

- VectorSlice :> VECTOR_SLICE

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

26.41 WORD

符号なし整数型のプリミティブ演算の提供.

```
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
```

シグネチャを実装するストラクチャ

- `Word : WORD`

  ```
  type word = word
  ```

  `Word32` は `Word` のストラクチャリプリケーション

- `Word64 : WORD`

  ```
  type word = word64
  ```

  `LargeWord` は `Word64` のストラクチャリプリケーション

- `Word8 : WORD`

  ```
  type word = word8
  ```

26.42 トップレベル環境

ライブラリで定義されているよく使う関数はトップレベルで再定義されている．本説では，Standard ML 標準ライブラリで定義されている型や関数の中でトップレベルで再定義されているものを示す．

- infix 宣言

  ```
  infix 7 * / div mod
  infix 6 + - ^
  infixr 5 :: @
  infix 4 = <> > >= < <=
  infix 3 := o
  infix 0 before
  ```

- type 宣言

  ```
  type substring = Substring.substring
  datatype order = datatype General.order
  ```

- exception 宣言

```
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 宣言

```
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
```

同値性チェックプリミティブ=と<>は，コンパイラが組込み関数として直接サポートしている．

- オーバロードされた識別子

```
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, stri
val <= : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32, str
val > : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32, stri
val >= : ['a::{int, word, int8, word8, int16, word16, int64, word64, intInf, real, real32, str
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]
```

第27章　SML#システムライブラリ

SML#は，Standard ML 標準ライブラリに加えて，SML#が提供する機能を使いこなすための SML#ライブラリを提供している．これらライブラリは，SML#固有のライブラリであるため，シグネチャは定義せず，直接インタフェイスファイルを通じて提供される．これらライブラリを構成するストラクチャは，ライブラリインタフェイスファイル名（⟨*librarySmiFilePath*⟩）で参照可能なように種類毎に分類され，`smi` ファイルが定義されている．

現在の版で提供しているライブラリは以下の通りである．

ライブラリ名	ストラクチャ名 (節)
`"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)

これらのライブラリは，SML#の拡張機能を実現するためにコンパイラから参照されることがある．分割コンパイルモードで以下の SML#拡張機能を用いる場合は，たとえライブラリ関数を使用していなかったとしても，インターフェースファイルから以下のライブラリを`_require` しなければならない．

- SQL 構文を用いる場合，`"sql.smi"`を`_require` しなければならない．
- `#reify` カインド付きの多相関数を呼び出す場合，`"reify.smi"`を`_require` しなければならない．

27.1　DynamicLink

インタフェイス

```
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
```

型

- lib オープンされた動的リンクライブラリのハンドルを表す抽象データ型.
- mode dlopen' でライブラリをオープンする際のモード. NOW は dlopen' 呼び出し時にオープンすることを意味する. LAZY は dlopen' 呼び出し時ライブラリファイルをチェックしオープンの準備をするのみで, ライブラリファイルは, 実行時に参照された時にオープンする処理を生成することを意味する.

27.2 Pointer

インタフェイス

```
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

SQL ライブラリの詳細は 22.8 節を参照せよ.

ネストしたストラクチャ

- Op(27.4)
- Numeric(27.5)

27.4 SQL.Op

SQL.Op ストラクチャの詳細は 22.9 節を参照せよ.

27.5 SQL.Numeric

SQL.Numeric ストラクチャの詳細は 22.10 節を参照せよ.

27.6 Pthread

インタフェイス

```
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

インタフェイス

```
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

インタフェイス

```
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
```

第28章　SML#コンパイラの起動

SML#コンパイラの起動は smlsharp コマンドによって行う．smlsharp コマンドは SML#コンパイラを起動し，SML#プログラムを機械語に翻訳し，オブジェクトファイルを生成し，リンクを行う．伝統的な C コンパイラドライバコマンドと同様に，ユーザーは smlsharp コマンドにモード選択オプションを指定することで，この一連のステップのうちの一部を実行することができる．また，SML#の対話環境も smlsharp コマンドから起動する．

smlsharp コマンドの書式は以下の通りである．

```
smlsharp [ option ··· ] [ -- ] [ inputFile ··· ]
```

smlsharp コマンドの引数は，コマンドラインオプションまたは入力ファイル名の列である．コマンドラインオプションはふつうマイナス記号（-）で始まる．コマンドラインオプションは-I や-L などの一部のオプションを除き順不同である．--をオプション列の終わりを明示するために用いても良い．--より後の引数はすべてオプションとして解釈されない．これら以外の引数は入力ファイル名である．入力ファイル名は，コマンドラインオプションの列の途中に現れてもよい．全てのオプションは入力ファイル名より先に解釈され，その効果は，全ての入力ファイルに対して適用される．

また，いくつかの環境変数は，smlsharp コマンドの動作に影響を与える．これらの環境変数は，SML#コンパイラがコンパイルしたプログラムの動作にも同様に影響する．

以下，smlsharp コマンドが解釈するコマンドラインオプションおよび環境変数を，オプションの種類に分けて列挙する．

28.1　モード選択オプション

以下のオプションは smlsharp コマンドの実行モードを指定する．smlsharp コマンドのコマンドラインには，以下のオプションのいずれかが高々1 つ指定されていなければならない．

--help ヘルプメッセージを表示して終了する．

-fsyntax-only 指定された.sml および.smi ファイルの構文検査のみを行い，コンパイルを中断する．後述する-o オプションをこのオプションと同時に指定することはできない．検査の結果はエラーメッセージと終了ステータスで通知される．

-ftypecheck-only 指定された.sml および.smi ファイルの型検査を行い，コンパイルを中断する．-o オプションをこのオプションと同時に指定することはできない．検査の結果はエラーメッセージと終了ステータスで通知される．

-S 指定された.sml ファイルをコンパイルし，アセンブリコードをファイルに出力する．出力ファイル名には，入力ファイル名の拡張子.sml を.s に置き換えた名前を用いる．入力ファイルがただ 1 つだけの場合に限り，-o オプションで出力ファイル名を指定することができる．

-c 指定された.sml ファイルをコンパイルし，オブジェクトファイルを出力する．出力ファイル名には，入力ファイル名の拡張子.sml を.o に置き換えた名前を用いる．入力ファイルがただ 1 つだけの場合に限り，-o オプションで出力ファイル名を指定することができる．

`-Mm` 入力ファイルとして指定された`.smi` ファイルをプログラムエントリとするプログラム全体をコンパイルおよびリンクするために必要な Makefile を生成する．指定された`.smi` ファイルから，コンパイルおよびリンクに必要な全てのファイルの依存関係が計算される．ml-lex や ml-yacc などのソースファイルを生成するツールを含まないプロジェクトならば，このモードはそのプロジェクトのための完全な Makefile を出力する．`-o` オプションと共に指定されている場合，標準出力の代わりに`-o` で指定されたファイルに結果を出力する．

`-MMm` `-Mm` と同じだが，標準ライブラリは出力から除外される．

`-M` 指定された`.sml` ファイルをコンパイルするときに読み込まれるファイルのリストを Makefile 形式で標準出力に表示する．`-o` オプションと共に指定されている場合，標準出力の代わりに`-o` で指定されたファイルに結果を出力する．

`-MM` `-M` と同じだが，標準ライブラリは出力から除外される．

`-Ml` 指定された`.smi` ファイルと指定してリンクするときにリンク対象となるオブジェクトファイルのリストを Makefile 形式で標準出力に表示する．`-o` オプションと共に指定されている場合，標準出力の代わりに`-o` で指定されたファイルに結果を出力する．

`-MMl` `-Ml` と同じだが，標準ライブラリは出力から除外される．

以上のいずれのオプションも指定されていない場合，入力ファイルの有無によって実行モードが異なる．

- 入力ファイルが指定されていない場合，対話環境を起動する．
- 入力ファイルが指定されている場合，`smlsharp` コマンドはリンクモードで起動される．入力ファイルのリストには，高々1 つの`.smi` ファイルまたは`.sml` ファイルを含めても良い．入力ファイルリストの中に`.smi` ファイルが指定されている場合，その`.smi` ファイルから`_require` 関係を辿って到達できる全ての`.smi` ファイルを列挙し，各`.smi` ファイルに対応するオブジェクトファイルを検索し，それらオブジェクトファイルをリンクの対象とする．オブジェクトファイルの検索については，後述する`-filemap` オプションの説明を参照せよ．入力ファイルリストの中に`.sml` ファイルが指定されている場合，その`.sml` ファイルをコンパイルした後，その`.sml` ファイルに対応する`.smi` ファイルからリンク対象のオブジェクトファイルのリストを計算する．高々1 つの`.smi` ファイルまたは`.sml` ファイルが含まれる．`.smi` または`.sml` 以外の入力ファイルは，システムリンカが受け付けるオブジェクトファイルやライブラリファイルを指定する．`smlsharp` コマンドは，これらオブジェクトファイルのリストと SML#ランタイムライブラリをシステムリンカコマンドを起動してリンクする．

  なお，システムリンカと同様，入力ファイルを指定する順番には意味がある．未参照シンボルを持つオブジェクトファイルは，そのシンボルを定義するオブジェクトファイルよりも前に指定されていなければならない．

  出力される実行形式ファイルのデフォルトの名前は `a.out` である．`-o` オプションによって出力ファイル名を指定することができる．

28.2　モード共通のオプション

`-o` *filename* 出力先ファイル名を指定する．ファイルに出力される内容は選択されたモードによって異なる．

`-v` `smlsharp` コマンドを verbose モードにする．主に `smlsharp` コマンドが実行する外部コマンドのコマンドラインとその出力が表示される．入力ファイルを伴わずに`-v` が指定された場合，コンパイラのバージョンを表示し終了する．

28.3 コンパイルオプション

以下のオプションはSML#コンパイラによるファイル検索およびコード生成を制御する．このカテゴリに属するオプションは対話環境やリンクモードにも作用する．

`-I`*dir* `.smi` ファイルの検索パスにディレクトリ *dir* を追加する．複数の`-I` オプションがコマンドラインに指定されている場合，指定された順番で各ディレクトリを検索する．なお，このオプションは，リンク時の`.smi` ファイルの検索にも使用される．

`-nostdpath` `.smi` ファイルの検索パスを`-I` で指定したディレクトリのみに限定する．

`-O`, `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Oz` 最適化レベルを指定する．`-O0` は全ての最適化を無効にする．`-O1` から`-O3` は数字が大きいほどより高度な最適化を有効にする．`-O` は`-O2` の別名である．`-Os` および`-Oz` はコードサイズを小さくする最適化を有効にする．これらのオプションがコマンドラインに複数指定された場合は，最後のひとつのみが有効となる．

`--target=`*target*, `-mcmodel` *model*, `-march` *arch*, `-mcpu` *cpu*, `-mattr` *attrs* LLVMのコード生成器に対して指定するターゲット，コードモデル，アーキテクチャ，CPU，およびコード生成属性を指定する．*model* は `small`, `medium`, `large`, `kernel` のいずれかである．*attrs* には属性のリストをコンマ区切りで指定する．*target*, *arch*, *cpu*, *attrs* に対して有効な指定はLLVM マニュアルあるいはLLVM付属の `llc` コマンドのヘルプを参照せよ．

`-fpic`, `-fPIC`, `-fno-pic`, `-mdynamic-no-pic` コードの再配置モデルを指定する．`-fpic` は`-fPIC` の別名である．`-fPIC` は再配置可能なコードを生成する．`-fno-pic` は再配置しない（non-relocatable な）コードを生成する．`-mdynamic-no-pic` は再配置可能な外部参照を含む再配置しないコードを生成する．これらのオプションが指定されていないときは，システム依存のデフォルトの再配置モデルが選択される．

`-Xllc` *arg*, `-Xopt` *arg* コード生成の際に実行されるLLVMの `llc` および `opt` コマンドに渡す追加の引数を指定する．複数の`-Xllc` や`-Xopt` がコマンドラインに指定された場合，全ての追加引数が指定された順にコマンドに渡される．

`-emit-llvm` コンパイル結果をファイルに書き出すとき，ネイティブコードの代わりにLLVM IRを出力する．`-emit-llvm` が`-S` と共に指定されているとき，アセンブリコードの代わりにテキスト形式のLLVM IR コードを出力する．デフォルトの出力ファイル名の拡張子は`.s` から`.ll` に変わる．`-c` と共に指定されているとき，オブジェクトファイルの代わりにLLVM bitcodeファイルを出力する．デフォルトの出力ファイル名の拡張子は`.o` から`.bc` に変わる．

28.4 リンクオプション

以下のオプションはリンカの振る舞いを制御する．このカテゴリに属するオプションは対話環境にも作用する．

`-l`*library* 指定された名前のライブラリをリンクする．対話環境で指定された場合，そのライブラリの関数を対話的にインポートすることが可能となる．

`-L`*dir* リンク時にライブラリを検索するパスにディレクトリ *dir* を追加する．複数の`-L` オプションがコマンドラインに指定されている場合，指定された順番で各ディレクトリを検索する．

`-nostdlib` ライブラリ検索パスを`-L` で指定したディレクトリのみに限定する．

`-c++` リンクコマンドとして C コンパイラドライバではなく C++コンパイラドライバを用いる．C++で書かれたオブジェクトファイルやライブラリを SML#プログラムにリンクする場合はこのオプションを指定する．

`-Wl` *args*, `-Xlinker` *arg* リンクの際にコンパイラドライバに追加で与える引数を指定する．`-Wl` の *args* にはコンマ区切りで引数のリストを指定する．`-Xlinker` はただひとつの引数を指定する．複数の`-Wl` および`-Xlinker` がコマンドラインに指定された場合，全ての追加引数が指定された順にリンクコマンドに渡される．

`-filemap` *filename* コンパイラがファイルを探すときに使用するファイル名の対応表を指定する．`smlsharp` コマンドは`.smi` ファイル名の拡張子を`.smi` から`.o` に置き換えた名前を対応するオブジェクトファイルの名前とする．このオプションを指定することで，この対応を変更することができる．*filename* の各行には，=記号，コンパイラが生成したファイル名，および本来アクセスすべきファイル名を，この順番で，スペースで区切って書く．*filename* には，SML#コンパイラが検索しようとする全てのファイルの名前が記載されていなければならない．*filename* に記載されていないファイルが必要になったときはエラーとなる．

28.5　対話モードのオプション

`-r` *smifile* 対話セッションの初期環境をセットアップするときに，指定された`.smi` ファイルを追加で読み込む．基本ライブラリに加えてユーザー自身のライブラリを対話環境で使いたいときに有用である．`.smi` ファイルが指し示すライブラリは，事前にコンパイルされ，全てのオブジェクトファイルが用意されていなければならない．`smlsharp` コマンドは，その`.smi` ファイルがリンクモードで指定されたかのようにオブジェクトファイルのリストを計算し，それらオブジェクトファイルを共有ライブラリとしてリンクし，システムの動的ロード機能を用いてロードする．

28.6　コンパイラ開発者向けオプション

以下のオプションは SML#コンパイラの開発用である．

`-d` [*switch*] SML#コンパイラ開発者向けスイッチ *switch* を指定する．*switch* が省略された場合，コンパイラ開発者向けの verbose モードを有効にする．*switch* のない`-d` が`--help` と共に指定された場合は，開発者向けスイッチの一覧を表示する．

`-B` *dir* SML#コンパイラの設定ファイル (config.mk) および標準ライブラリが存在するディレクトリを指定する．

28.7　環境変数

SML#コンパイラは以下の環境変数を参照する．

`SMLSHARP_HEAPSIZE` 最小および最大ヒープサイズのヒントを，この順番でコロンで区切って並べ，バイト単位で指定する．ヒープサイズには接尾辞 `K`（キロ），`M`（メガ），`G`（ギガ），`T`（テラ）を付けることができる．最大ヒープサイズは省略することができる．その場合，ヒープの動的な拡張は行われない．デフォルトは “`32M:256M`” である．もしメモリ不足でコンパイラが中断したならば，この環境変数を設定しヒープサイズを増やしてみると良い．

`SMLSHARP_VERBOSE` ランタイムの verbose モードを 0 から 5 の整数で指定する．デフォルトは 2 である．SML#コンパイラの開発用である．

SMLSHARP_LIBMYSQLCLIENT, SMLSHARP_LIBODBC, SMLSHARP_LIBPQ SML#が動的リンクするデータベースライブラリのファイル名を指定する．SML#のデータベース機能を使用したときにライブラリのロードエラーが発生する場合，これらの環境変数を設定する．

28.8 典型的な使用例

28.8.1 対話環境

対話環境を起動する最も簡単な方法は，引数なしで smlsharp コマンドを実行することである．

```
$ smlsharp
```

SML#の対話環境からある C ライブラリの関数を対話的に利用したい場合，そのライブラリの名前を-l オプションで指定する．例えば，対話環境から zlib ライブラリを呼び出したいときは以下のコマンドを実行する．

```
$ smlsharp -lz
```

C 標準ライブラリや POSIX スレッドライブラリなど，SML#コンパイラがデフォルトでリンクするライブラリは-l で指定しなくても利用できる．

28.8.2 プログラムのコンパイル

ファイルに書かれた SML#プログラムをコンパイルする場合，もしただひとつの.sml ファイル（および対応する.smi ファイル）のみから成るプログラムならば，その.sml ファイルをただ指定するだけで実行形式ファイルを得ることができる．

```
$ smlsharp foo.sml
```

実行形式ファイルの名前を a.out から変えたいときは-o オプションを指定する．

```
$ smlsharp -o foo foo.sml
```

28.8.3 分割コンパイルとリンク

複数のファイルから成る SML#プログラムを手でコンパイルしてリンクするときの標準的な手順は以下の通りである．

```
$ smlsharp -c -O2 foo.sml
$ smlsharp -c -O2 bar.sml
$ smlsharp -c -O2 baz.sml
$ smlsharp foo.smi
```

まず，SML#プログラムを構成する各.sml ファイルを-c を付けた smlsharp コマンドでそれぞれ独立かつ順不同にコンパイルする．foo.sml のコンパイルに成功すると，オブジェクトファイル foo.o が生成される．全ての.sml ファイルのコンパイルが完了した後，プログラム全体のエントリとなるコードを持つ.sml ファイルのインターフェースファイル（ここでは foo.smi）を smlsharp コマンドに指定し，プログラム全体をリンクする．このとき smlsharp コマンドは，foo.smi から_require 宣言で辿れる.smi ファイルを全て列挙し，拡張子を.o に変え，リンク対象のオブジェクトファイルのリストを得る．

ソースツリーの管理の都合などにより.smi ファイルから.o ファイルへの対応を制御したい場合は，以下の例のように，ファイル名の対応表を持つファイルを作り-filemap オプションで指定する．

```
$ 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
```

このとき，objmap の内容は以下の通りである．

```
= foo.o obj/foo.o
= bar.o obj/bar.o
= baz.o obj/baz.o
```

SML#プログラムは任意の C/C++プログラムやライブラリとリンクすることができる．SML#プログラムを C/C++ライブラリとリンクする場合は，リンク時の smlsharp コマンドラインにリンクする C/C++のオブジェクトファイルやライブラリファイルを列挙する．例えば，独自の C プログラム util.c と OpenGL を利用する SML#プログラム foo.sml をコンパイル・リンクする場合は以下のようにする．

```
$ smlsharp -c -O2 foo.sml
$ cc -c -O2 util.c
$ smlsharp foo.smi util.o -lgl -lglu
```

28.8.4　Makefile の生成

分割コンパイルを使いこなすには，コンパイルとリンクを自動化する “make” コマンドと組み合わせると良い．make コマンドを使うためには，ファイル間の依存関係が書かれた Makefile を作る必要がある．smlsharp -Mm コマンドを使うことで，SML#プログラムを make でビルドするために必要な完全な Makefile を自動的に作ることができる．

例えば，3 つの.smi ファイル foo.smi，bar.smi，および baz.smi（とそれらに対応する.sml ファイル）からなるプロジェクトがあるとする．これらの間には以下の_require 関係がある．

- foo.smi は bar.smi を_require している．
- bar.smi は baz.smi を_require している．

このとき，

```
$ smlsharp -MMm foo.smi -o Makefile
```

は以下の Makefile を生成する．

```
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
```

あとはただ make を実行するだけでビルドが完了する．

```
% 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
```

プログラムを変更して _require 関係が変化したならば，その都度 smlsharp -MMm コマンドで Makefile を生成し直す必要がある．

もしあなたのプロジェクトに ml-lex や ml-yacc などのソースファイルが含まれているならば，smlsharp -MMm だけでは十分ではない．それらのツールを起動して .sml ファイルを生成するルールを追加する必要がある．例えば，構文解析器 parser.grm を含むならば，smlsharp -Mm が生成した Makefile に対して，以下のような parser.grm.sml を生成するための規則を手で追加する必要がある．

```
parser.grm.sml parser.grm.sig: parser.grm
        ml-yacc parser.grm
parser.grm.sig: parser.grm.sml
```

smlsharp -MMm が生成する Makefile と，これら追加のルールが書かれた Makefile を分けておくと便利であろう．そのためのひとつの方法は，Makefile には追加のルールと自動生成ファイルの include を書くことである．例えば，Makefile に以下のように書く．

```
# デフォルトのターゲットは all である
all:

# 依存関係は自動生成して取り込む
depend.mk:
        smlsharp -Mm foo.smi -o depend.mk
include depend.mk

# 追加の依存関係
parser.grm.sml parser.grm.sig: parser.grm
        ml-yacc parser.grm
parser.grm.sig: parser.grm.sml
```

もちろん，あなたのお気に入りの Makefile テクニックを駆使してもよい．

第29章　SML#実行時データ管理

SML#は，全ての基本的なデータ構造の実行時表現として，C 言語の一般的な処理系と同様に，ターゲットプラットフォームにおいて最も自然なデータ表現（あるいは，ターゲットプラットフォームの ABI（Application Binary Interface）が規程するデータ表現）を採用している．例えば，x86_64 プラットフォームでは，`int` 型は 32 ビット整数の型，`real` 型は 64 ビットの IEEE754 浮動小数点数の型である．これらの型の式の評価は，それぞれの型の値を計算するのに最も適したレジスタを用いて行われる．また，これらの型が組や配列の要素であった場合，それぞれの型に適した境界条件で要素が整列される．このような自然なデータ表現は，多相関数を含む，SML#プログラム全体で維持される．

SML#の分割コンパイルおよび C 言語との相互運用性は，この自然な実行時データ表現を基礎として構築されている．従って，これらの機能を用いるユーザーは，SML#の実行時データ表現についての知識が必要である．以下，本章では，SML#の実行時データ表現とメモリ管理について詳述する．

29.1　実行時表現

SML#が管理する全ての型構成子は，メモリ上での「サイズ」，GC のトレース対象かどうかを表す「タグ」，およびメモリ上のビット列の意味および取り得る範囲を表す「表現」の 3 つの要素を持つ．この 3 つ組を，その型構成子の「実行時表現」と呼ぶ．型の実行時表現は，型を構成する最も外側の型構成子によって決定される．主要な組み込み型構成子の実行時表現は以下の通りである．

組み込み型構成子	サイズ（バイト数）	タグ	表現
int, int32	4	0	C の符号付き整数の内部表現
int8	1	0	C の符号付き整数の内部表現
int16	2	0	C の符号付き整数の内部表現
int64	8	0	C の符号付き整数の内部表現
word, word32	4	0	C の符号なし整数の内部表現
word8	1	0	C の符号なし整数の内部表現
word16	2	0	C の符号なし整数の内部表現
word64	8	0	C の符号なし整数の内部表現
char	1	0	不定
real, real64	8	0	IEEE754 浮動小数点数
real32	4	0	IEEE754 浮動小数点数
ptr	4 または 8	0	C の void*の内部表現
codeptr	4 または 8	0	C の関数ポインタの内部表現
unit	4	0	単一バリアント
contag	4	0	不定
boxed	4 または 8	1	不定
array	4 または 8	1	NULL でないポインタ
vector	4 または 8	1	NULL でないポインタ
ref	4 または 8	1	NULL でないポインタ
string	4 または 8	1	NULL でないポインタ
exn	4 または 8	1	NULL でないポインタ
レコード型，組型	4 または 8	1	NULL でないポインタ
関数型	4 または 8	1	NULL でないポインタ

サイズが「4 または 8」である型のサイズは，ターゲットプラットフォームが 32 ビットか 64 ビットかに依存して決まる．

すべての型構成子について，境界整列条件はサイズに等しい．C 言語の内部表現と等しい実行時表現は，その C 言語の内部表現と等しい大きさと境界整列条件を持つことが期待されている一方，現在の SML#コンパイラは，各実行時表現の大きさと境界整列条件をターゲットによらず固定で与えている（この点は SML#の将来の版で改良する予定である）．

datatype 構文で定義されたユーザー定義型の実行時表現は，ユーザーが与えたコンストラクタの集合から計算される．

- 引数を取らないただ一つのデータコンストラクタから構成されるユーザー定義型の実行時表現は unit に等しい．
- 引数を取らない 2 つ以上のデータコンストラクタのみから構成されるユーザー定義型の実行時表現は contag に等しい．
- それ以外のユーザー定義型の実行時表現は boxed に等しい．

type 宣言で定義された型の別名は，型の実行時表現を計算する際は全て展開される．不透明シグネチャ制約によって導入された不透明型の実行時表現は，その不透明型の実装型の実行時表現に等しい．インターフェースファイルによって不透明型として宣言された型の実行時表現は，その不透明型宣言で与えられた実装型の実行時表現を持つ．

29.2 ガーベジコレクションの影響

タグが 1 である型のデータは，SML#のメモリ管理機構の影響を受ける．以下，これらのデータを boxed データと呼ぶ．boxed データを保持するメモリ領域は，そのデータのコンストラクタが評価されたときに暗黙に割り当てられ，SML#のガーベジコレクションによって自動的に解放される．boxed データを C 関数に渡す場合，そのデータの内容だけでなく，そのデータを保持するメモリ領域が解放されるタイミングにも注意する必要がある．boxed データは，C プログラムから見て以下の性質を持つ．

1. boxed データは，C の `malloc` 関数で確保したメモリ領域と同様に，一度メモリ上に確保されると，解放されるまでそのアドレスは変化しない．従って，SML#の配列や `ref` など，SML#プログラムから見て一意性を持つ書き換え可能なデータは，C 関数から見ても同一の一意性を持つ．組型などもメモリ上を移動しないため，一度 C 関数が組型の boxed データへのポインタを取得したならば，その boxed データが生きている限り，そのデータへのポインタは不変である．

2. SML#プログラムから C 関数を呼び出す際に引数として渡された boxed データは，原則としてそのまま C 関数に渡される．C 関数呼び出しに伴うデータの変換，再確保，再生成などは行われない．従って，`array` や `ref` などの書き換え可能なデータを C 関数で書き換えた場合，その書き換えの効果を SML#プログラムから観測することができる．

3. C 関数の引数として渡された boxed データは，呼び出した C 関数の実行が終了するまでの間は解放されない．C 関数の実行が終了し SML#プログラムに制御が戻ったとき，その boxed データが SML#プログラムから参照可能でなければ，その boxed データは解放される．

4. C 関数にコールバック関数として渡された SML#の関数は，プログラムの終了まで解放されない．従って，C 関数は，受け取った SML#関数への関数ポインタをグローバル変数などに保存し，関数呼び出しを超えて呼び出すことができる．また，異なるスレッドからコールバック関数が呼び出されたとしても，コールバック関数の自由変数の値は必ず保存されている．一方，この仕様のため，SML#プログラムでは，コールバック関数を取る C 関数の呼び出しがメモリリークを起こす可能性があることに注意しなければならない．SML#コンパイラは，トップレベルで定義された関数のみをコールバック関数として使用する場合，メモリリークが起こらないことを保証する．

5. SML#のガベージコレクタは正確なガベージコレクタである．boxed データへのポインタが C のヒープ領域に保存され，C プログラムからそのデータが参照可能であったとしても，そのデータが SML#プログラムから参照できなければ，ガーベジコレクタはその boxed データを解放する．そのため，C 関数は，SML#プログラムから受け取った boxed データへのポインタを，グローバル変数や `malloc` 関数で確保した領域などに保存し，SML#からの C 関数呼び出しを越えて boxed データへのポインタを保持してはならない．なお，上述の通り，C 関数が受け取った boxed データおよびそこから到達可能な全ての boxed データは，その C 関数の実行が終了するまでの間は移動も解放もされないため，boxed データへのポインタをレジスタやスタックフレームに保存することは問題なく可能である．

29.3 スタックを巻き戻すジャンプの影響

SML#および C/C++は，通常のコール/リターンだけでなく，関数の呼び出しスタックを巻き戻し，リターン先以外の場所にジャンプする機能を備えている．SML#および C++では例外，C では `setjmp` および `longjmp` がこれに該当する．コールバックを含む C 関数をインポートした SML#プログラムにおいても，これらのジャンプ機構を，その機構に期待する通りに利用することができる．SML#の例外機構および C/C++のジャンプ機構による SML#プログラムへの影響を以下に列挙する．

1. SML#の例外機構は，C++の例外機構と同様，Itanium ABI を通じて実装されている．従って，同じ Itanium ABI で実装された例外機構の cleanup ハンドラは，SML#の例外によっても実行される．

例えば，C++のローカル変数に対するデストラクタは，SML#のコールバック関数を呼び出した際に例外が発生した場合でも，適切に実行される．

2. SML#の `raise` 式で発生した例外は，SML#の `handle` 式でのみ捕捉できる．`handle` 式は SML#の例外のみを捕捉し，その他の巻き戻しジャンプは捕捉できない．

3. SML#の例外がどの `handle` 式にも捕捉されなかった場合，プログラム全体は中断され，システムに異常終了を報告する．

4. C++の例外と SML#の例外は，呼び出しスタックに SML#のコールバック関数が含まれていたとしても，互いに独立に，期待通りに動作する．

5. C の `setjmp` および `longjmp` も，SML#プログラムから見て単なる巻き戻しジャンプとして使用されている場合に限り，呼び出しスタックに SML#のコールバック関数を挟んでいたとしても，期待通りに動作する．`setjmp` および `longjmp` 関数を用いて SML#プログラム内にループを作ることはできない．

29.4　マルチスレッドの影響

SML#では，スレッドを生成する任意の C 関数を SML#プログラムにインポートすることができる．C 関数によって新たに生成されたスレッドから SML#のコールバック関数によって呼び出されたとしても，SML#ランタイムは新たなスレッドの生成を検知し，SML#プログラムを複数のスレッドで実行する．このとき，複数のスレッドによる SML#プログラムの実行は，C プログラムと同様，オペレーティングシステムのマルチスレッドサポートにより，異なるスレッドで並行に実行される．もしオペレーティングシステムがマルチコア CPU 上での複数スレッドの同時実行をサポートしているならば，SML#プログラムのスレッドも同時に実行される．

複数の SML#プログラムのスレッドは，単一のヒープ領域を共有する．従って，SML#の配列などの書き換え可能なデータ構造を利用することで，ヒープを介したスレッド間通信を行うことができる．スレッド間で共有されるメモリ領域の排他制御は，排他制御を行う C 関数をインポートし呼び出すことで行う．SML#はヒープに確保したデータの移動を行わないため，SML#のヒープ上に，セマフォなどの排他制御のためのデータ構造を確保することができる．

以上が，SML#が提供する最も基本的なマルチスレッドサポートである．SML#は標準では高水準なスレッドライブラリを提供しないが，ユーザーはこのマルチスレッドサポートを利用し，より高水準な並行プログラミングサポートを提供するライブラリを SML#で書くことができる．SML#に適した，より高水準な並列プログラミングフレームワークは，将来の版の SML#で提供する予定である．

第IV部

プログラミングツール

第30章　構文解析器生成ツール smlyacc と smllex

SML#には構文解析器生成ツール smlyacc が同梱されている．このプログラムは，SML/NJ110.73 用に Andrew W. Appel と David R. Tarditi Jr. によって開発されたプログラムを SML#に移植したものである．オリジナルなソフトウェアのライセンス情報は，smlyacc のソースディレクトリ `src/ml-yacc/COPYRIGHT` に，また，その使い方は `src/ml-yacc/doc/mlyacc.pdf` にある．文法規則の書き方等は，ドキュメントを参照のこと．ここでは，smlyacc と smllex を SML#とともに使用する方法を概説する．

30.1　生成されるソースファイル

SML#システムをインストースするとコマンド `smlyacc` と `smllex` が同時にインストールされる．これらは SML#コンパイラを生成するためにも使用される．これらコマンドは，以下のファイルを生成する．

コマンド	入力ファイル	生成ファイル	内容
`smlyacc`	⟨*YaccInputFileName*⟩`.grm`	⟨*YaccInputFileName*⟩`.grm.sml`	構文解析プログラム
		⟨*YaccInputFileName*⟩`.grm.sig`	パーザトークンシグネチャ
		⟨*YaccInputFileName*⟩`.grm.desc`	LR オートマトン状態記述
`smllex`	⟨*LexInputFileName*⟩`.lex`	⟨*LexInputFileName*⟩`.lex.sml`	字句解析プログラム

- `smlyacc` コマンドには入力ファイル名入力ファイル名 ⟨*YaccInputFileName*⟩.grm のみを指定する．出力ファイル名を変更したい場合は，以下の環境変数で設定できる．

  `SMLYACC_OUTPUT=`⟨*YaccOutputFileName*⟩*.sml*

  この指定をすると，トークンシグネチャは，指定したファイルの先頭に出力される．出力ファイル名の指定はサフィックス（`.sml`）も含め指定する．

- `smllex` コマンドには入力ファイル名入力ファイル名 ⟨*LexInputFileName*⟩.lex のみを指定する．出力ファイル名を変更したい場合は，以下の環境変数で設定できる．

  `SMLLEX_OUTPUT=`⟨*LexOutputFileName*⟩*.sml*

  出力ファイル名の指定はサフィックス（`.sml`）も含め指定する．

30.2　smlyacc 入力ファイルの構造

`smlyacc` の入力ファイルは以下の形式で記述する．

⟨ ユーザ定義の *SML#*コード ⟩
%%
⟨ 構文規則解析のための *YACC* 宣言 ⟩
%%
⟨ 構文規則とその属性の定義 ⟩

1. ユーザ定義の SML#コードのセクションには，構文規則の属性定義等で使用する補助関数などのユーザコードを指定する．

2. 構文規則解析のための YACC 宣言では，非終端記号の結合方向などの構文定義の解釈に関する宣言や smlyacc の動作に関する指示を指定する．非終端記号の結合方向などの構文定義の解釈に関する宣言に関しては標準の YACC の入力形式と同一である．詳しくは，src/ml-yacc/doc/mlyacc.pdf を参照せよ．

   SML#で使用する場合，smlyacc が生成するストラクチャに関する以下の指示を指定する．

```
%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 指定．パーザの名前を指定する．この名前からトークンストラクの名前<Name>_TOKENS が生成される．
   - %header 指定．smlyacc は，パーザ本体をストラクチャのボディとして以下の形式で生成する．

```
= struct
    ...
end
```

     %header() はこのストラクチャ本体に前置するコードフラグメントをカッコの中に指定する．パーザを SML#の分割コンパイルモードで使用する場合，単独のストラクチャとなるように，上記のようにストラクチャ宣言を指定する．
   - %pos 指定．smllex で使用するポジションの型を指定する．
   - %eop 指定．構文解析を終了する終端記号集合を指定する．
   - %term 指定．終端記号名の集合を SML#の datatype のコンストラクタの指定の形式で定義する．ここで指定されたそれぞれの終端器号名に対して，EOF : pos * pos -> token （引数が無い場合）や CHAR : char * pos * pos -> token （引数がある場合）の型のトークン生成関数が作成される．これら関数はトークンストラクチャ<Name>_TOKENS にまとめられ smllex と共有される．
   - %nonterm 指定．非終端記号名の集合とその非終端記号を左辺にもつ規則が還元された場合にそれに付随して生成される属性の型を指定する．

3. 構文規則とその属性の定義では，終端記号と非終端記号の定義おおび構文解析の対象言語の LALR 文法とその属性を規則の集合として記述する．これら規則の文法は標準の YACC の入力形式と同一である．詳しくは，src/ml-yacc/doc/mlyacc.pdf を参照せよ．

30.3 smlyacc 出力ファイルの構造とインタフェイスファイル記述

smlyacc が生成する ⟨*YaccInputFile*⟩.grm.sml は，以下のシグネチャをもつストラクチャである．

```
signature ML_LRVALS =
  sig
     structure Tokens : ML_TOKENS
     structure Parser : PARSER
     sharing type Parser.token = Tokens.token
  end
```

このシグネチャを含む smlyacc のサポートライブラリは，smlyacc-lib.smi にまとめられている．生成された構文解析プログラムを SML#で使用するためには，以下のインタフェイスを書く必要がある．

```
_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
```

Parser ストラクチャの pos，arg，result 型以外は，そのまま指定する．Token ストラクチャは，smlyacc が生成する ⟨*YaccInputFileName*⟩.grm.sig の内容をここにコピーする．

それぞれの要素の概要を以下に説明する．

- token 型．smllex で生成される字句解析処理が返す語彙（トークン）を表す抽象データ型．
- stream 型．smlyacc で生成される構文解析器の入力ストリーム型．
- result 型．構文解析器の出力である抽象構文木のデータ型．⟨*YaccInputFile*⟩.grm で指定した最上位の構文規則の属の性の型と同一である．
- pos 型．⟨*YaccInputFile*⟩.grm で指定し，smllex で生成した字句解析器が使用するポジション型．
- arg 型．⟨*YaccInputFile*⟩.grm で指定したパーザの引数型．必要なければ unit 型でよい．
- ParseError 例外．構文解析器が構文エラーを検出した時の例外．
- makeStream 関数．構文解析器の引数に指定する stream 型を生成する関数．引数には，通常，smllex が生成する字句解析器を指定する．
- consStream 関数．現在の入力ストリームに字句を一つ戻す関数．
- getStream 関数．現在の入力ストリームを読み先頭の字句を一返す関数．
- sameToken 関数．字句の同一性をチェックする関数．
- parse 関数．構文解析関数．現在の入力を指定して呼び出すと，構文解析結果と残りのトークンストリームを返す．

30.4　smllex 入力ファイルの構造

smllex の入力ファイルは以下の形式で記述する．

⟨ ユーザ定義 ⟩
%%
⟨ 字句解析のための *LEX* 宣言 ⟩
%%
⟨ 正規表現とその属性の定義 ⟩

1. ユーザ定義セクション 字句解析処理が使用する型の定義，正規表現の属性定義等で使用する補助関数などのユーザコードを指定する．

   以下の 2 つは必ず指定しなければならない．

   ```
   type lexresult = ...
   fun eof () = ...
   ```

   - lexresult 型．字句解析器が正規言語を受理した時に返す値の型を指定する．smlyacc と共に使用する場合は，通常，⟨*YaccInputFile*⟩.grm ファイルで定義する token 型である．
   - eof 関数．構文解析器がファイル終了を検出した時に呼び出される関数．lexresult 型の値で end of file を表現する値を返す．

2. LEX 宣言セクション

   smllex の動作に関する指示や，正規言語を定義するための補助定義などを宣言する．正規言語を定義するための補助定義は，通常の LEX システムと同様である．

   smllex の動作に関する指示には以下のものがある．

- %structure declareation．smllex は，字句解析プログラムを一つのストラクチャとして生成する．本宣言では，そのストラクチャの名前を以下のように指定する．

  ```
  %structure MLLex
  ```

- %arg declareation．smllex が生成する字句改正プログラムに与えるユーザ引数の型を指定する．ユーザ引数が必要ない場合は省略可能である．
- %full declareation．8bit 文字を使用することを宣言する．日本語を扱う場合はこの指定をする必要がある．

3. 正規言語セクション 認識すべき字句の集合を正規表現で指定する．

指定の詳細は，src/ml-lex/doc/mllex.pdf を参照．

30.5 smllex 出力ファイルのインタフェイスファイル記述

smllex は，字句解析生成関数 makeLexer を含むストラクチャを生成する．生成された字句解析プログラムを SML#で使用するためには，そのインタフェイスを書く必要がある．最小のインタフェイスは以下の形である．

```
_require "basis.smi"
_require "coreML.grm.smi"

structure CoreMLLex =
struct
  val makeLexer : (int -> string) -> unit -> CoreML.Tokens.token
end
```

makeLexer の第一引数は，最大文字数を引数としてとり文字入力列を返す関数である．この関数を，makeLexer (fn n => Text
のように文字列入力に適用すると，呼び出される毎に字句を返す字句解析関数が生成される．smlyacc によって生成された makeStream 関数をこの関数に適用すると，構文解析器が使用するトークンストリーを生成できる．

追加引数等のユーザ宣言は UserDeclarations ストラクチャに定義される．それらを使う字句解析プログラムのインタフェイスファイルは以下の形である．

```
_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
```

第Ⅴ部

SML#の内部構造

第31章　序文

第V部では，SML#コンパイラの内部構造を詳述する．本部の目的は，本文書の第II部などを習得しML系関数型言語の素養を持つ者がSML#ソースコードの詳細を理解することである．読者としては，SML#コンパイラの開発に加わろうとする者やSML#ソースコードを新しいコンパイラ開発等に利用しようとする者を主に想定しているが，高水準プログラミング言語のコンパイラに興味を持つ一般の読者にも参考となる文書となるように配慮した．

オープンソースソフトウェアには尊敬すべき優れたコードが数多く存在する．それらの文化に接して筆者が感じる問題点の一つは，それら優れたコードの構造や機能を，他の開発者や興味ある読者に理解できるように文書化する努力が往々にして欠如していることである．コードそれ自体がドキュメントであるとの主張は数万行を超える大規模システムに対しては現実的ではない．大規模システムでは，それぞれのコード断片自身では理解不可能な大域的な仮定や，他の複数のコードを制御するためのデータが含まれる．それらを理解するには，関連するシステム全般に関する処理の流れと，その実現のためにシステムの各部分に分散してコード化されたデータの意味を把握する必要がある．現時点での唯一の方法は文書化であると考える．さらに，コードの意図や構造を記述した文書は，それ自身，新たな構造や機能の示唆を与えうる財産となると期待される．

そこで，本文書では，その範を，筆者が尊敬するVAX/VMS OSの内部構造の詳細な記述文書 [4] に取り，SML#コードが扱うデータ構造とコードの処理の詳細を，各機能が基礎とする理論やアルゴルズムと共に記述することにした．

第V部の構成は以下の通りである．

1. 第32章ではSML#ソースパッケージの構成を記述する．
2. 第33章ではSML#コンパイラの制御の流れの概要を記述する．
3. 以降の各章では，第33章で記述したコンパラの制御の流れに従い，各モジュールの構造と機能を記述する．第**??**章では，`main`モジュールを記述する．
4. 第**??**章は`toplebel`モジュールを記述する．
5. 順次執筆しUPLOADいたします…
6.

第32章　SML#ソースパッケージ

SML#配布パッケージ smlsharp-3.7.1.tar.gz は，SML#コンパイラ，基本ライブラリ，サポートツールのソースを含む.

32.1　ソースパッケージの構成

SML#のソースディストリビューション smlsharp-3.7.1.tar.gz には以下のディレクトリとファイルが含まれる.

- ディレクトリ

benchmark/	ベンチマークソース
precompiled/	minismlsharp のアセンブリソースアーカイブ
sample/	サンプルプログラム
src/	SML# ソースファイル
test/	テストプログラムとデータ

- ファイル

Changes	リリース情報ファイル
INSTALL	インストール手順
LICENSE	SML#ライセンス
Makefile.in	Makefile ファイルテンプレート
RELEASE_DATE	リリース版日付
VERSION	バージョン
config.h.in	config.h ファイルテンプレート
config.mk.in	Makefile の動作を制御する config.mk ファイルテンプレート
configure.ac	autoconf への入力ファイル
depend.mk	ソースファイルの依存関係
files.mk	ファイル集合定数の定義ファイル
mkdepend	depend.mk 作成スクリプト
precompile.mk	precompiled/再構築用 make スクリプト

32.2　SML# ソースツリー

SML#のソースディレクトリは，以下のディスディレクトリとファイルが含まれる.

- デイレクトリ
  - SML#コンパイラソースコード

basis/	Standard ML 基本ライブラリ
compiler/	SML#コンパイラ
config/	configure が設定するシステムパラメタアクセスライブラリ
ffi/	C 言語直接連携サポートライブラリ
llvm/	LLVM コード生成ライブラリ
runtime/	実行時処理系
sql/	SQL 統合サポートライブラリ
thread/	スレッドサポートライブラリ
unix-utils/	UNIX 基本コマンドライブラリ

– SML#ツール

smlformat/	smlformat 清書プログラム生成器
smlunit/	単体テストライブラリ

これらは，SML#のために開発した汎用性あるツールである．これらは，比較的小規模な独立したシステムであり，本解説ではこれらの記述は省略する．

– サードパーテイのコード

ml-lex/	字句解析器生成ツール
ml-yacc/	構文解析器生成ツール
smlnj/	basis/で使用する smlnj ソースファイル
smlnj-lib/	smlnj ユーテティリティライブラリ

これらサードパーテイのコードは SML#をビルドするために，SML#用に移植されたものである．make システムによって SML#ビルド時にコンパイルされ使用されるため，それぞれのライセンスとともにソースコードがここに置かれている．これらコードは本解説の対象としない．

- ファイル

basis.smi	Standard ML 基本ライブラリのインタフェイスファイル
builtin.smi	組込みデータを束縛するインタフェイスファイル
config.mk	コンパイラコマンドが使用する環境変数定義
config.mk.in	config.mk のテンプレート
config.sed	config.mk 生成のための sed スクリプト
ffi.smi	C との直接連携ライブラリのインタフェイスファイル
foreach.smi	並列機能のインタフェイスファイル
json.smi	JSON サポート機能のインタフェイスファイル
ml-yacc-lib.smi	smlyacc ライブラリのインタフェイスファイル
prelude.smi	対話型環境のインタフェイスファイル
reifyTy.smi	自己反映計算機能のインタフェイスファイル
smlformat-lib.smi	smlformat ライブラリのインタフェイスファイル
smlnj-lib.smi	smlnj-lib のインタフェイスファイル
smlunit-lib.smi	smlunit のインタフェイスファイル
sql.smi	SQL 統合サポートライブラリのインタフェイスファイル
thread.smi	スレッドサポートライブラリのインタフェイスファイル

32.3 compiler ディレクトリ

これらの中で compiler ディレクトリはさらに階層化されている．

- ディレクトリ

- compilePhases/ : コンパイルフェーズ
  - bitmapcompilation/ ビットマップ生成
  - cconvcompile/ コーリングコンベンションコンパイル
  - closureconversion/ クロージャ変換
  - datatypecompilation/ データ型コンパイル
  - elaborate/ 構文論的評価
  - fficompilation/ C言語連携コンパイル
  - llvmemit/ LLVM コードエミッション
  - llvmgen/ LLVM コード生成
  - loadfile/
  - machinecodegen/ 低レベルコード生成
  - main/
  - matchcompilation/ パターンマッチングコンパイル
  - nameevaluation/ 名前評価とモジュールコンパイル
  - parser/ 構文解析
  - recordcalcoptimization/ 型付きレコード計算最適化処理
  - recordcompilation/ 型主導レコードコンパイル
  - stackallocation/ スタックフレーム割り当て
  - toplevel/
  - typedcalcoptimization/ 型付き中間言語最適化
  - typedelaboration/
  - typeinference/ 型推論，カリー関数最適化
  - valrecoptimization/ 相互再帰的関数最適化処理
- compilerIRs/ : コンパイラ中間表現
  - absyn/ 抽象構文木
  - anormal/ A-normal 中間言語
  - bitmapcalc/ ビットマップを明示した中間言語
  - closurecalc/ クロージャ生成を明示した中間言語
  - idcalc/ スコープ規則を持たない中間言語
  - llvmir/ LLVM 中間表現
  - machinecode/ レジスタトランスファ言語
  - patterncalc/ 型無し中間表現
  - recordcalc/ 多相型レコード計算
  - runtimecalc/ 低レベル中間言語
  - typedcalc/ 型付き中間言語
  - typedlambda/ 型付きラムダ計算
- data/ : 型や定数などのデータ
  - builtin/ コンパイラ組込みデータ
  - constantterm/ 定数定義
  - control/ コンパイラの動作パラメタ
  - name/ 実行コードラベル
  - runtimetypes/ 実行時型
  - symbols/ 変数，ラベル等の表現
  - types/ 型表現
- extensions/ : 種々のコンパイル機能

concurrencysupport/	並列スレッドサポート
debug/	デバッグ機能
foreach/	超並列_foreach 構文サポート
format-utils/	smlforma 用フォーマッタライブラリ
json/	JSON サポート
reflection/	コンパイル時のレフレクションサポート
usererror/	コンパイラエラー処理
userlevelprimitive/	ユーザコードによるコンパイラ拡張機能

– libs/ : コンパイラが使用するライブラリ

digest/	& SHA ハッシュライブラリ
env/	辞書ユーテリティ
heapdump/	実行時ヒープイメージダンプ関数
ids/	カウンタユーテリティ
interactivePrinter/	対話型プリンタ
list-utils/	リストユーティリティ
toolchain/	UNIX toolchain コマンド
util/	各種ユーティリティ関数

- ファイル

minismlsharp.smi	minismlsharp のインタフェイスファイル
minismlsharp.sml	smlsharp のコンパイル用コンパイラトップレベル
smlsharp.smi	smlsharp のインタフェイスファイル
smlsharp.sml	smlsharp トップレベル

compiler のサブディレクトリを含む各ディレクトリは，main サブディレクトリを含み，このディレクトリ以下にソースファイルが置かれている．従って，例えば抽象構文木のソースファイルは compilerIRs/absyn/main/ 下に置かれている．ソースファイルには，拡張子.sml を持つプログラムファイルと拡張子.smi を持つ同名のインターフェイスファイルが含まれる．

これに加えて，以下の拡張子を持つファイルは，ソースファイルを生成ツールの入力ファイルである．

.ppg	プリンターコード自動生成器 smlformat の入力ファイル．
.grm	smlyacc の入力ファイル．
.lex	smllex の入力ファイル．

これらファイル名に.smi が付加したインターフェイスファイルは，自動生成されたソースファイルのインタフェイス記述である．

32.4 basis ディレクトリ

basis/ライブラリは基本ライブラリのソースファイルであり，その main サブディレクトリは以下のファイルを含む．

1. シグネチャファイル

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. 共通コード

ArraySlice_common.sml
Array_common.sml
VectorSlice_common.sml
Vector_common.sml

3. ストラクチャファイルディレクトリ

   以下の各名前に.sml を付加したプログラムファイルおよび.smi を付加したインタフェイスファイルが含まれる.

   ```
   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#サポートファイル

   以下は，各ストラクチャファイルがその効率よい実装を実現するために使用する SML#の低レベルなサポート関数ファイルである.

SMLSharp_Runtime	SML#実行時プリミティブ
SMLSharp_OSFileSys	OS ストラクチャ用プリミティブ
SMLSharp_OSIO	IO ストラクチャ用プリミティブ
SMLSharp_OSProcess	OS.Process 用プリミティブ
SMLSharp_RealClass	Real ストラクチャ用プリミティブ
SMLSharp_ScanChar	scan プリミティブ

5. トップレベル

toplevel.sml	トップレベル定義
toplevel.smi	トップレベル定義インターフェイスファイル

第33章　コンパイラの制御構造

本章ではSML#コンパイラの制御の流れを記述する．

1. ソースロケーション
   - src/compiler/minismlsharp.{smi,sml} ファイル src/compiler/smlsharp.{smi,sml} ファイル
   - src/compiler/compilePhase/main ディレクトリ
   - src/compiler/compilePhase/toplevel ディレクトリ
2. 概要
   (a) src/compiler/minismlsharp.sml によるコンパイラのスタートアップ．
   (b) src/compiler/compilePhase/main モジュールによるコンパイラコマンドのメイン処理．
   (c) src/compiler/compilePhase/toplevel モジュールによるトップレベルのコンパイル処理．

33.1　コンパイラスタートアップ

SML#コンパイラコマンドは，SML#言語で書かれたプログラムを SML#コンパラで分割コンパイル・リンクし生成された実行形式プログラムである．第??章で記述する通り，SML#のプログラムのメインコードは，SML#コマンドをリンクモードで起動した時に指定されたインタフェイスファイルに対応するソースファイルおよび，そのインタフェイスファイルから参照されるインタフェイスファイルに対応するソースファイル集合のトップレベルの宣言の列を，依存関係に従い順次実行するコードである．

SML#コンパイラをリンクするコードは Makefile に src/compiler/smlsharp をターゲットとして記述されている以下の shell コマンドである．

```
        $(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 $@
```

標準の環境では，SMLSHARP_STAGE1 は minismlsharp コマンドと定義されている．minismlsharp は，SML#コンパイラのソースコードをコンパイルするのに必要な最小のライブラリリンクされた smlsharp コンパイラである．smlsharp コンパイラのトップレベルソースファイルはこのコマンドで指定されたルートインタフェイスファイル./src/compiler/smlsharp.smi に対応する以下のファイルである．

```
    ./src/compiler/smlsharp.sml
```

このソースファイルの内容は以下の４行である．

```
val commandLineName = CommandLine.name ()
val commandArgs = CommandLine.arguments ()
val status = Main.main (commandLineName, commandArgs)
val () = OS.Process.exit status
```

Main.main は comiler/compilePhases/main/main にある Main.sml ファイルに書かれた以下の型を持つ main 関数である．

```
val main : string * string list -> OS.Process.status
```

この関数が SML#コンパイラのトップレベル関数である．この関数は，コマンド名（smlsharp）とコマンドライン引数文字列リストを受け取り，コンパイルやリンク処理を実行する．

33.2　コンパイラコマンドのメイン処理

SML#コンパイラの初期化処理等を行うメイン関数は，comiler/compilePhases/main/main/の Main.sml ファイルに書かれたの型を持つ main 関数である．

```
val main : string * string list -> OS.Process.status
```

この関数は，コマンド名と引数リストを受け取り，command 関数を呼び出し，以下の処理を行う．

1. 引数を解釈し，コンパイラの動作モードと動作オプションを決定する．
2. 動作モードに対応した以下のいずれかの処理を実行する．
   (a) ソースファイルリストのコンパイル
   (b) ソースファイルのコンパイル及びリンク
   (c) インタフェイスファイルで指定されたシステムのリンク
   (d) 依存関係ファイルの出力と Makefile の生成
   (e) 種々の情報のプリント

   ソースファイルのコンパイルは，compileSMLFile 関数によって以下の手順で行われる．

   - ソースファイルをオープン
   - コンパイルモードを決定し，command 関数で用意されたコンパイル環境 topContext を生成する．
   - Top.compile を呼び出す．Top.compile 関数は，comiler/compilePhases/top/main/の Top.sml ファイルに書かれたの型を持つ関数である．

     ```
     val compile
         : LLVMUtils.compile_options
           -> options
           -> toplevelContext
           -> Parser.input
           -> InterfaceName.dependency * result
     ```

     返される値の型 InterfacceName.dependency はソースファイルが依存するインタフェイスファイルのリスト，result はコンパイル結果のオブジェクトファイルである．

   コンパイル及びリンクは，link 関数によって以下の手順で行われる．

   (a) 入力ソースファイルが.sml ファイルなら，コンパイルしオブジェクトファイルを生成し，ルートのオブジェクトファイルとする．入力ソースファイルが.smi ファイルなら，ファイル名から，ルートのオブジェクトファイルを名を決定する．
   (b) 入力ソースファイルが.sml ファイルなら，コンパイル関数が返す依存するインタフェイスファイルのリストから，リンクするオブジェクトファイル集合を決定する．入力ソースファイルが.smi ファイルなら，ファイルを loadSMI 関数で解析し，トップレベルのソースが依存するインタフェイスファイルのリストをもとめ，リンクするオブジェクトファイル集合を決定する．

(c) ルートオブジェクトファイル，依存オブジェクトファイル，ライブラリリストを指定しシステムリンカ呼び出しリンクを実行する

33.3 コンパイラのトップレベル

SML#コンパイラのトップレベル関数は，comiler/compilePhases/toplevel/main/の Top.sml ファイルに書かれた以下の型を持つ compile 関数である．

```
val compile :
    : LLVMUtils.compile_options
      -> options
      -> toplevelContext
      -> Parser.input
      -> InterfaceName.dependency * result
```

この関数は，LLVM コード生成オプション，コンパイルオプション，コンパイル環境，ソースファイルのインプットストリームを受け取り，ソースファイルをコンパイルし，ソースファイルが依存するインタフェイスファイルのリスト（InterfaceName.dependency）とコンパイル結果のオブジェクトファイル（result）を返す．

この関数は，Main.sml の compileSML 関数から呼び出され，以下の表に示す通り，各コンパイルフェーズの関数を順に呼び出しソースファイルのコンパイルを行う．

Compile step	Compile Phase Function	Source Language	Target Language
1	Parser.parse	(input stream)	Absyn.absyn
2	LoadFile.load	Absyn.absyn	AbsynInterface
3	Elaborator.elaborate	AbsynInterface.compile_unit	PatternCalcInte
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.tpdec
8	UncurryFundecl.optimize	TypedCalc.tpdecl list	TypedCalc.tpdec
9	PolyTyElimination.compile	TypedCalc.tpdecl list	TypedCalc.tpdec
10	TPOptimize.optimize	TypedCalc.tpdecl list	TypedCalc.tpdec
11	MatchCompiler.compile	TypedCalc.tpdecl list	RecordCalc.rcde
12	FFICompilation.compile	RecordCalc.rcdecl list	RecordCalc.rcde
13	RecordCompilation.compile	RecordCalc.rcdecl list	RecordCalc.rcde
13	DatatypeCompilation.compile	RecordCalc.rcdecl list	TypedLambda.tld
14	BitmapCompilation2.compile	TypedCalc.tpdecl list	BitmapCalc2.bcd
15	ClosureConversion2.convert	BitmapCalc2.bcdecl list	ClosureCalc.pro
16	CallingConventionCompile.compile	ClosureCalc.program	RuntimeCalc.pro
17	ANormalize.compile	RuntimeCalc.program	ANormal.program
18	MachineCodeGen.compile	ANormal.program	MachineCode.pro
18	ConcurrencySupport.insertCheckGC	MachineCode.program	MachineCode.pro
19	StackAllocation.compile	MachineCode.program	MachineCode.pro
19	LLVMGen.compile	MachineCode.program	LLVMIR.program
20	LLVMEmit.emit	LLVMIR.program	LLVM.LLVMModule

第 VI 部

参考文献，その他

関連図書

[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.