修 士 論 文

マルチスレッド型プロセッサによる関数型言語の実行方式に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

Jin Yu

2004 年 3 月

修 士 論 文

マルチスレッド型プロセッサによる関数型言語の実行方式に関する研究

指導教官　日比野 靖 教授

審査委員主査　日比野 靖 教授
審査委員　田中 清史 助教授
審査委員　堀口 進 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

210027　Jin Yu

提出年月: 2004 年 2 月

概 要

本論文では、マルチスレッド型プロセッサアーキテクチャと関数型言語の特徴組み合わせることにより、関数型言語の高効率実行を目的とする。マルチスレッド型プロセッサは、プロセッサの各パイプラインステージをすべて異なるスレッドからの命令で埋める機構により、高い命令スループットを実現できる。関数型言語はその言語特有の性質から実行効率は逐次型アーキテクチャマシンではあまり良くない。効率良い実行を妨げる原因としては主に２つある。まず、プログラム中で頻繁に起きる関数の呼び出しにより、通常のパイプランではハザードを生じる。マルチスレッド型プロセッサの導入によって、ハザードを回避する。次に、プログラムに内在する並列実行可能性は無視されている。代表的な関数型言語である SML のコンパイラは主に型推論を行い、並列性の検出を行っていない。この研究では、SML に着目し、並列性検出とスレッド間の同期を行うコンパイラを提案する。

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　研究の背景

高級言語計算機は現在では顧みられなくなり、通常命令のパイプライン実行をするマイクロプロセッサ時代を迎えている。高級言語計算機はソフトウェアの課題をハードウェアで解決するという試みであった。高級計算機の存在意義としては以下のことが考えられた。

1. 通常マシンのコンパイラが複雑になり過ぎた。
2. プログラミングが容易になり、ソフトウェアの開発費用を削減する。
3. これまでソフトウェアで処理してきたことを安価になるハードウェアで処理することで上昇するソフトウェア費用を削減する。
4. 計算機システムは高級言語を効率良く実行できるように設計されるべきである。
5. コード能率が良い。
6. 高級言語計算機こそが高級言語を効率良く実行できる費用効率の良いアーキテクチャである。

しかし、これらはすべて誤りであった。上記に関しては以下のように考えるのが妥当である。

1. コンパイラが複雑になるのは、コード生成が困難だからではなく、前処理、語彙解析、構文処理、最適化、誤り検出、誤り回復等の部分が相当の規模になっているからである。
2. マシンの水準によらず高級言語プログラムの効率の良いデバッグ環境を開発するべきである。
3. ソフトウェアアルゴリズムをハードウェア化するための設計費用、設計時間は膨大であり、ソフトウェアの開発費用を償えない。また、ソフトウェアはその複製費用がただ同然であるのに対して、ハードウェアの複製費用 (LSI チップの製造費用) は集積技術がいかに進んだとしてもその低下には限りがある。
4. 第一に、高級言語指向ということは、対象の言語を一つに限るという危険を犯すことであり、第二に言語実行の効率化を目標とするあまり、システムの総合的効率を無視しがちであるということである。
5. コード能力が良いことと、性能が良いこととは一致しない。しかもコード効率が達成されるか疑問である。

6. 仮に直接実行マシンが実現できたとしても、デバッグ済みのプログラムを、実行の度に構文解析/誤り診断を受けるのは全くの無駄であり、構文解析/誤り診断は、コンパイラ時に行い、実行時は通常レベルのマシンコードを実行した方が遥かに費用効率が良い。

つまり、計算機システムのユーザ評価は、高級言語計算機の達成度ではないのである。総合的な費用効率の良さにある。以上のことから、高級言語計算機の存在意義は否定され、高級言語計算機という捉え方での研究は行われなくなり、同時に高級プログラム言語の研究も低調となった。しかしながら、ソフトウェアの課題すなわちソフトウェアの開発費用がますます増大していくという問題は未解決のままであった。ソフトウェアの課題としては主に以下の二つが考えられる。

1. ソフトウェア開発の方法を工学的に捉えようとするソフトウェア工学という課題
2. プログラミングのパラダイムを根本から見直し、これまでのノイマン型アーキテクチャに基づく命令型のパラダイムから、関数型、論理型、オブジェクト指向型といったそれまで一般的でなかったプログラムパラダイムに転換を図るという課題

この中でも関数型プログラミング、あるいは論理型プログラミングは、並列処理あるいは並列計算機アーキテクチャの観点から関心がもたれた。しかしながら、これらも総合的な費用効率の良さとは一致しなかった。費用効率という観点からは、より簡単な構成、より少ない資源で実現されるマシンの方が、性能で若干劣っていたとしても優位に立つ。従って、高級言語計算機は成立し得ない。
一般的にプログラム言語の処理は、以下の三つに分けられる。

1. 言語の表層的な構文の処理
2. 演算実行の順序制御
3. 演算実行

このうち、第一の構文の処理はコンパイラに任せるのが費用効率の点から優れていることは上記で述べた。一方、ハードウェアによる高速化に最も効果があるのが演算実行部である。ハードウェアによる高速化は並列動作による効果である。高級言語処理に関して残された部分は、演算実行の制御である。これに対しては現在は命令パイプラインを有する逐次型アーキテクチャマシンを対象した命令生成と命令スケジューリングにのみ関心を払われた。関数型、論理型、オブジェク指向型は逐次型アーキテクチャでシミュレートできるが、これらのパラダイムに内在する並列実行可能性は無視されている [2]。並列論理型言語に焦点をあわせ、遅い段階での型判定をするための 1 命令で多方向への分岐可能なタグ付き分岐命令をマルチスレッド型プロセッサに適した研究が行われている [16]。本研究では関数型言語の実行に必須なスタック操作を効率よく実行するためのマルチスレッド型プロセッサを採用する共に、多数スレッドの生成と同期を行うためのコンパイラ方式を提案する。

1.2 研究の目的

本研究は、マルチスレッド型プロセッサアーキテクチャと関数型言語の特徴を組み合わせることにより、関数型言語の高効率実行を目的とする。マルチスレッド型プロセッサは、プロセッサの各パイプランステージをすべて異なるスレッドからの命令で埋める機構により、高い命令スループットを実現できる。関数型言語はその言語特有の性質から実行効率は逐次型アーキテクチャマシンではあまり良くない。効率良い実行を妨げる原因としては主に２つある。

まず、プログラム中で頻繁に起きる関数の呼び出しによる、スタック操作の命令の実行である。この操作は通常のパイプラインではハザードを生じる。マルチスレッド型プロセッサの導入によって、ハザードを回避する。

次に、プログラムに内在する並列実行可能性は無視されている。厳密な意味での関数型言語は副作用を持たない。プログラムはある関数 f で表され、プログラムの実行は入力 x に対する f の **適用**$f(x)$ を評価することと定義される。関数 f の適用には入力として、別の関数や f 自身が用いられる。例えば、$f(x) = g(h(x), k(x))$ のような場合、この評価は、一番外側の g の評価から行うこともできれば、中の $h(x)$ や $k(x)$ の評価から行うこともでき、$h(x)$ と $k(x)$ を同時に評価することもできる。厳密な関数言語では、これらの異なる評価順序のいずれを用いても評価結果は同じであることが保証されている。この性質は並列処理の可能性を示している。代表的な関数型言語 SML のコンパイラは主に型推論を行っているが、逐次型マシンを交換としているので並列性の検出を行っていない。

この研究では、並列性検出とスレッド間の同期を行うコンパイラを提案する。

1.3 本論文の構成

以下、本論文では次のような構成に従って、マルチスレッド型プロセッサによる関数型言語の実行方式に関する研究について論じる。

まず第二章では、マルチスレッド型のプロセッサを紹介することにより、その特徴を述べ、何故関数型言語の実行にふさわしいかを説明する。次に第三章では、関数型言語がどのようなものであり、またどのように処理されるかを述べる。この中で、関数型言語の特徴を述べた後、関数型言語のパイプライン実行上での問題点を述べる。ひいては、言語に内在する並列性を記述し、主に Standard ML 言語に焦点をあてて説明をする。第四章では、本論文で主題である、並列性の検出とスレッド生成が可能なコンパイラの実装を行う。典型的な関数型言語として Standard ML に着目し、言語のサーブセットを取って、コンパイラの実装する。たとえば、型は int だけして、その上でのいろいろな関数の定義、関数の適用、式の評価等を考える。また第五章では、提案したコンパイラの優位性を評価する。最後に第六章では、本研究をまとめるともに、今後の課題について概観する。

第2章　マルチスレッド型プロセッサアーキテクチャ

2.1　通常のパイプラインプロセッサアーキテクチャ

現在の一般的な計算機の多くがパイプライン処理を行っている。しかし、単一の命令ストリームをパイプライン処理する場合、パイプラインハザードの問題が避けられない。図2.1はパイプラインを示す。この図2.1に示すように、命令1と命令2の間に依存関係が存在すると（図の中では依存関係を矢印で表している)、この依存関係が解消されるまでは命令2はパイプライン中でストールすることになる。このことによって、命令1と命令2の間にパイプラインバブルが発生する。

構造ハザード(資源競合)を避けるために複数の機能ユニットを用意したり、演算パイプラインを採用し、データハザードや制御ハザード(分岐や割り込み)を避けるために静的または動的スケジューリングを行っても、ハザードを完全に回避することはできない。

動的スケジューリングや命令の並列実行のための複雑な制御を持つ高度なスーパースカラプロセッサが多く現れたが、競争力としてTime to Marketが重視されLSI設計の短期間化が要求されるという流れの中、複雑な制御を伴うアーキテクチャは設計の複雑化により動作周波数の向上や機能追加などへの対処といった面で設計上または製造上の問題が大きい。CMOSテクノロジの微細化に伴う配線遅延やクロックスキーの相対的増加などを考えると、将来の製造プロセスで非常に複雑なプロセッサを設計するのはより困難になる[17]。

そこで考えられたのが、ハザードのないパイプラインを持ち、複雑な制御なしで複数のスレッドを並列に実行するマルチスレッド型プロセッサである。

2.2　マルチスレッド型プロセッサアーキテクチャの特徴

マルチスレッド型プロセッサは、パイプライン段数と同数の独立な命令流(スレッド)を並列に実行し、図2.2すべてのステージを独立なスレッドの命令(つまりデータ依存、制御依存関係のない命令)で埋め、さらにスレッド数と同数のレジスタセットを用意することで、通常のパイプライン処理で問題となるデータ依存、制御依存、資源競合をなくし、パイプラインハザードを完全に回避する。１つのスレッドだけに注目すればパイプライン化されない逐次処理であるため通常のパイプライン処理よりもレイテンシは大きいが、複

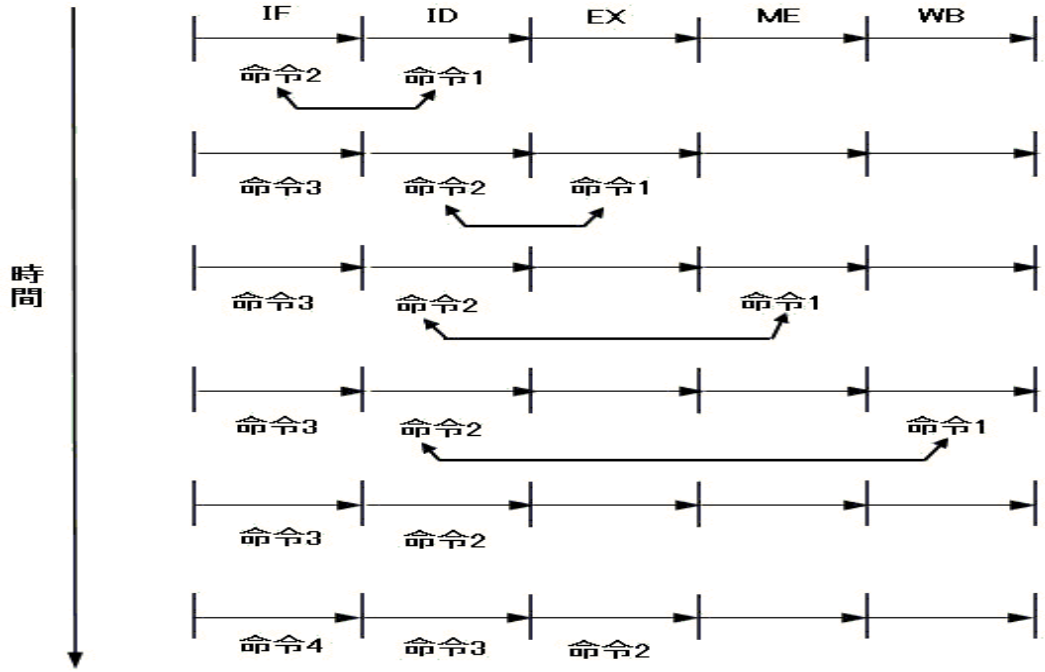

図 2.1: 通常のパイプラインバブル

数のスレッドを並列かつストールなしに走らせることができるのでシステム全体のスループットという意味での処理能力は向上する [1]。

マルチスレッド型プロセッサの考え方自体は古くからあるが、あまり普及することはなかった。しかし近年になって、IBM の 2 スレッド版 PowerPC や Tera MTA などの商用ベースのプロセッサが発表され、将来の有望なアーキテクチャの一つとして注目されるようになった。

以下にその特徴を示す。

- ハザードが発生せずパイプラインはストールしない。よって各パイプラインステージのリソース使用効率は非常に高くなる。
- 独立な複数個のスレッドを並列に実行する。実装するスレッド数によるが、マルチプログラミング環境、マルチメディアやネットワーク関係の性質などを考えると並列に実行可能なスレッドは十分に存在すると考えられる。
- スレッドの切り替えをハードウェア的に行うので、割り込みによるソフトウェア的なコンテクストスイッチよりもオーバーヘッドが格段に小さい。
- 毎サイクルのメモリ参照を可能にするには、十分な容量を持つパイプラインキャッシュが必要になる。
- メモリレイテンシが許容され(パイプライン化により隠蔽できる)、プロセッサを busy に保つことができる。

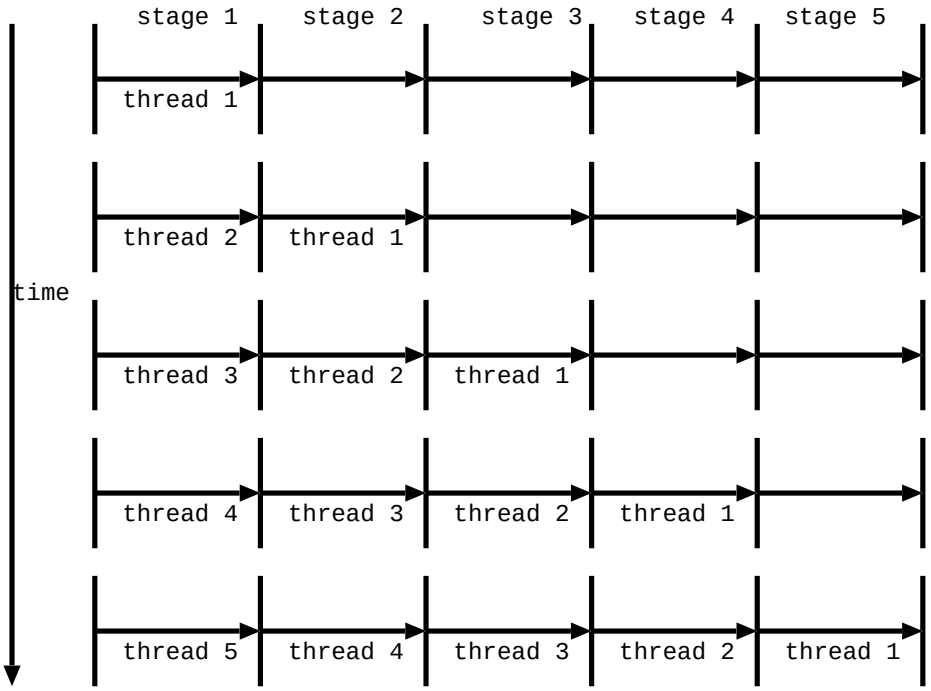

図 2.2: マルチスレッド型プロセッサのパイプライン

- 同時に処理可能なスレッドが十分に存在しないと高いスループットが得られない。最悪の場合、パイプライン化されない単一プロセッサと同程度になってしまう。
- 一つのスレッドだけに着目するとパイプライン化されない逐次処理であるためフォワーディングの必要がなく、パイプラインにフィードバックがない。つまりデータはパイプラインを一方向にしか流れない。
- 一部を除いてレジスタはスレッド数個分用意されるのでスレッド間でアクセス競合が生じない。

複数のスレッドが同時に走るので十分なバンド幅を提供するメモリ構成が重要である。また、並列に実行可能なスレッドの抽出、プロセッサを遊ばないようなスレッドの入れ換えなど、並列検出や複数のスレッドの生成が可能なコンパイラが性能を大きく左右する。

第3章　関数型言語の特徴とその処理方式

3.1　関数型言語の概要

本研究では特定のプログラミング言語に向いたプロセッサのアーキテクチャを論ずる。そのためには、対象となる言語の持つ性質や特徴を知らなければならない。特に専用プロセッサの最大の特徴である高速処理の実現のためには、プログラムがどのように処理され、またどのような操作が頻繁に実行されるのかを、具体的明らかにする必要がある。

そこで本章では、関数型言語の仕様や特徴、更には具体的な処理の方法について代表的な関数型言語である Standard ML を対象に論ずる。ML は、エジンバラ大学で 1970 年代の後半に、証明導出システム Edinburagh LCF のための記述言語として開発された。LCF システムの記述の対象は、定理や公理、推論規則などであり、ML は、それらを計算機システムの中で表現、操作するための言語、すなわちそれら数学的対象が属する形式言語のメタ言語 (Meta Language) であった [3]。その後、いくつか改良され、いくつかのバージョンができた。

- エジンバラ大学による EDML
- Standard ML of New Jersey
- CAML (Caml,Caml-light,Caml-super-light など)

これらは、基本的な考え方は同じだが、開発者はまったく別である。本研究では、Standard ML を使う。SML は、

- 変数の静的スコープ規則
- 型推論
- コンパイラ方式

という特徴を持つ。また、プログラムの意味を厳密に定義することができる。特に、型推論 (type inferences) は

$$\left\{ \begin{array}{l} \text{項（プログラミング言語では式と同じ）の型を宣言} \\ \text{すること無しで、処理系が自動的に型を推論する。} \end{array} \right\}$$

というものである。この機構は、最近ではいくつかの言語に備わっているが SML は、その中で最も有名なものと言える。多くの ML の処理系は、初期の所では LISP の上で構築されたが、EDML、SML は C 言語を用いて効率よいものとなっている。また、コンパイラ方式でありながら、そのインターフェースはインタプリタ方式と同様であり、簡単にプログラミングすることができる。

以下の本章では、まず Standard ML のプログラミング言語としての構造を概説した後、その特徴の処理について述べる。本研究ではコンパイラの実装において、言語のサーブセットを取って実装するので、言語の全部ではなくサブセットに用いる文法だけ着目する。

3.1.1 関数型言語の特徴

関数型プログラム言語によるプログラミングは、数々の定義を行うことと計算機を用いて式を評価することによる。プログラマの重要な役割は与えられた問題を解く関数を構築することである。この関数は補助的な関数を含むこともあるが、通常の数学の原則に従った表記法で表現される。計算機の役割は評価、すなわち計算する当事者であることであり、式を評価してその結果を表示するのが仕事である。この点では計算機は卓上計算器と同じような働きをする。関数計算機として際立った能力を発揮できるかどうかは計算能力を高めるように定義を行うプログラマの腕にかかっている。プログラマの定義した関数名が現れる式は、与えられた定義を単純化（すなわち、簡約化、reduction）の規則として用いて評価されて表示できる形に変換される。

関数型プログラミングの１つの特徴は、式がきちん確認した値を持つならば、計算機による評価結果は評価する順序によらないということである。すなわち、式の意味はその値であり、計算機の仕事は単にその値を求めることなのである。したがって、関数プログラム言語の式は、数学の式と同様に代数的な法則を用いて構築したり、操作したり、推論に用いたりすることができる。その結果として、単純で、簡潔で、柔軟性に富み、プログラミングにおける概念上の強力な構成法となることができる [7]。

関数型プログラミングは、与えられた問題を解くために、関数を定義し、それに引数を適用して値を得ることが基本である。プログラムは、数学的な関数と代数の概念に基づいて構成される。従来の手続き型言語のような [**代入**] がなく、変数は値を表す名前であり、メモリセルの参照ではない。まだ、繰り返しなどの制御文がなく、再帰的関数とかパターンマチングによって定義される。

そのいくつかの特徴を以下に説明する [3],[4],[6]。

変数の静的スコープ規則

上での解説した通り、ML の変数は値に付けられた名前であり、代入などの操作により変更することはできない。しかし、別な値を持つ同一の名前を何度でも定義し直すことができる。ML プログラムを正しく理解するためには、変数の有効範囲を理解しておく必要

がある。変数定義の有効範囲を変数のスコープと呼ぶ。

変数の宣言には、変数を値や関数に束縛する **val** 宣言や **fun** 宣言、および関数の仮引数宣言がある。

一つの **val** 宣言の一般的な形は、以下のようにいくつかの変数の定義を **and** で結んだものである。

$$
\begin{aligned}
&\textbf{val } x_1 = exp_1 \\
&\textbf{and } x_2 = exp_2 \\
&\quad \vdots \\
&\textbf{and } x_n = exp_n
\end{aligned}
$$

最後の ; はなくてもよい。この宣言によって変数 $x_1, \cdots, x_n$ が定義される。

以下説明の都合上、宣言の列を $decls, decls_i$ のように書き、さらに今注目している **val** 宣言を **VAL** とする。**VAL** で宣言される変数のスコープは、**VAL** の現れる場所に応じて以下のように定められている。

1. **VAL** がトップレベルでの宣言の場合、それに続くすべての部分の中で、同一の変数が再定義された部分を除く部分。

2. **let** $decls_1$ **VAL** $decl_2$ **in** exp **end** の場合、$decl_2$ および exp の中で、同一の変数が再定義された部分を除く部分。

3. **local** $decl_1$ **VAL** $decl_2$ **in** $decl_3$ **end** の場合、$decl_2$ および $decl_3$ の中で、同一の変数が再定義された部分を除く部分。

この規則により、複数の変数を同時に定義し直すことが可能である。

fun 宣言の一般的な形は、以下のようにいくつかの関数の定義を **and** で結んだものである。

$$
\begin{aligned}
&\textbf{fun } f_1\ p_1^1 \cdots p_{k1}^1\ = exp_1 \\
&\textbf{fun } f_2\ p_1^2 \cdots p_{k2}^2\ = exp_2 \\
&\quad \vdots \\
&\textbf{fun } f_n\ p_1^n \cdots p_{kn}^n\ = exp_n;
\end{aligned}
$$

この宣言によって、関数名と各関数の仮引数に含まれる変数が定義される。各仮引数 p_i^k に含まれる変数のスコープは、対応する関数本体 exp_k の中で同一の名前が再定義された部分を除く部分である。関数名 $f_1, \cdots, f_n$ のそれぞれのスコープは、上の **val** 宣言における規則で定められるスコープに、この宣言の本体のすべての式 $exp_1, \cdots, exp_n$ を加えたものである。

以上の規則で明示的に述べられているように、同一の名前が再定義されると、その新しい定義のスコープでは、以前の定義が隠され、新しい定義が有効になる。しかし、変数の再定義は、同一の名前に対する以前の定義を変更したり無効にしたりするものではなく、以前の定義を使用しているプログラムの意味は変化しない。例えば、次は再定義の簡単な例である。

- val x =20;
val x = 20 : int
- val y = 2 * x;
val y = 40 : int
- val x = 300;
val x = 300 : int
- y;
val it = 40 : int

このプログラムにおいて、x が再定義された後も y の値は変化しない。このように、静的スコープを持つ言語では、変数の値は、その変数が定義された時点の値に固定され、以降変化することはない。この規則は関数定義においても適用される。すなわち、関数の本体が仮引数以外の変数を含む場合、それらの変数の意味は、関数が定義された時点でのその変数に束縛されている値であり、関数が実行される時点での同一名の変数の定義とは関係がない。

これを理解するために、次の例を見てみよう。

- val x = 3;
val x = 3 : int
- val y = 5;
val y = 5 : int
- fun f x = x * y;
val f = fn : int → int
- f 5;
val it = 25 : int
- val y =1000;
val y = 1000 : int
- f 5;
val it = 25 : int

上の例の関数定義 $fun\ f\ x\ =\ x\ *\ y$ における x は仮引数宣言であり、それまでに定義されていた $val\ x\ =\ 3$ を隠し、後に関数適用によって与えられた実引数に束縛される。

一方この関数の中の y は、この関数が定義された時点での y の値を意味し、この関数が適用される時点で y が再定義されていても、変化しない。したがって、関数適用 $f\ 5$ は同一の値を計算する。

高階関数、参照透明性

関数プログラミングの考え方として特徴的なことは、関数を第１級の対象とするということである。手続き型言語の多くのものが関数や手続きを整数などとは異なる対象として扱っているが、関数プログラミング、ML では関数そのものをデータと同格に扱う。このことは関数を引数したり、関数を結果として返したりする関数を定義できる。関数を操作する関数を高階の関数と呼ぶ。ML において高度で簡潔なプログラムを書くための１つの鍵は、高階の関数を用いて関数を値として扱う技法に習熟することである。

以下に簡単な例を示す。

まず、関数を返すの関数の例—として、n^m を計算する関数を考えてみる。

```
- fun power m n = if m = 0 then 1
                  else n * power (m-1) n ;
val power = fn : int→ int →  int
- power 3 2;
val it = 8 : int
- val cube = power 3;
val cube = fn : int → int
```

上の定義によって、power が、power m n として使う高階の関数として定義され、「m を受取り、任意の n に対して n^m を計算する関数を返す関数」である。更に、power 3 を評価することによって cube は整数を 3 乗する関数が得られる。

次に、関数を引数として受取りそれを使って処理を行う高階の関数も定義できる。例えば、関数 f（定義済み関数だとする。）を引数として受取り、n に関する関数 $\sum_{k=1}^{n} f(k)$ を返す高階の関数 summation は以下のように定義できる。

```
- fun summation f n = if n =1 then f 1
                      else f n + summation f (n-1);
val summation  =  fn : (int → int) → int → int
```

関数プログラムの性質で最も重要なものは、プログラムの中に書かれている式の意味が、その文脈のみによって定まり、式の評価過程には依存しないという参照透明性（referential transparency）である。参照透明性は関数プログラムの変換や合成などプログラム開発に

対する方法論の基礎になっているものである、このことが関数プログラミングをほかの（手続きなど）プログラミングに対して際立たせている点である。

高階関数や参照透明性とは別の観点からも関数プログラミングの特徴をあげることができる。関数型言語には基本概念は極めて少ないということである。手続き型言語を学ぶ際の障害の１つは代入の概念である。また、同時に式、文、データなどの多様な計算対象を理解しなくてはならないということである。その一方で、関数プログラミングには基本的には関数抽象と関数適用しか存在しない。すなわち、抽象化と具体化しか必要としないのである。実用的な観点からは、数のような基本的なデータや非破壊的なリストなどのデータ構造を導入したり、計算対象の命名機能（宣言）を用意することが多いが、基本的な概念が少ないことはプログラミングを極めて理解しやすいものにしている。

3.1.2 関数型言語処理方式

対話型プログラミング

ML のプログラムは一つの式である。複雑で大きなプログラムも、多数の式の組み合わせによって構成される一つの式である。プログラムの実行は式が示す値を計算することによって行われる。ML でのプログラミングは、Lisp などの対話型言語同様、以下の手順を繰り返すことによって行われる。

1. ユーザによるプログラミングの入力。
2. システムによる型検査、プログラムのコンパイラと実行。
3. 実行結果とその型情報の表示。

以下に簡単な対話型プログラミングの例を示す。

```
- 345;
val it = 345 : int
- (20 - 10) * 43 + 89;
val it = 519 : int
- val name = "jin yu";
val name = "jin yu" : string
-"name: " ^ name;
val it = "name: jin yu" : string
-
```

ユーザの入力行の先頭の “-” は、ML システムが表示する入力促進、最後の “;” は入力の終りの指示である。val x = E は、式 E の結果に x という名前をつけ、後の式で使用するための構文である。

実用的な Standard ML の処理系は、インタープリタではなくコンパイラとして実装されており、入力されたプログラムは機械語に翻訳され、対話型システムに含まれる実行時カーネルの下で直接実行される。このため、対話型システムありながら、高い実行効率が得られる。

翻訳されたプログラムを、対話型システムとは独立に実行可能なプログラムにすることもできる。さらに多くの処理系では、モジュール単位の分割コンパイラを行う機能が提供されている。

計算の副作用

ノイマン型アーキテクチャでは、プログラムは CPU に対する指令を１列に並べたものであり、CPU はこの指令を一つずつ順にメモリより読み出して、その指令に従ってメモリのある番地に格納されているデータを読み出し、指定された演算を施し、その結果をメモリのある番地に書き込む。メモリの内容がマシンの状態に対応し、計算は状態を次々に変えることにより実行される。このように、計算は副作用を用いて実行される。

このような計算モデルは、命令型（impreative）であるという。FORTRAN,ALGOL, PL/I,PASCAL,C などのプログラム言語は、この計算モデルに準拠しており、命令型言語と呼ばれる。このような言語では、副作用は変数への代入を指示する代入文で表現される。

関数の値は、入力変数の値によって一意的に定まる。つまり、副作用がないので、関数の実行は互いに独立となり、並列処理が可能である。

3.1.3 関数型言語のパイプライン実行上の問題点

スタックフレームとフレームポインタ

１つのスタックを用いて、作業領域、戻り番地、引数の情報、局所変数、外側のスコープの情報、関数の呼び出しに伴って破壊されると困るレジスタ内容の写しなどを各関数の呼び出しごとに1組にまとめてスタック上に割り当てる。これをスタックフレーム（Stack Frame）または駆動レコード (Activation Record) と呼ぶ。スタックフレームの形を図 3.1 に示す。

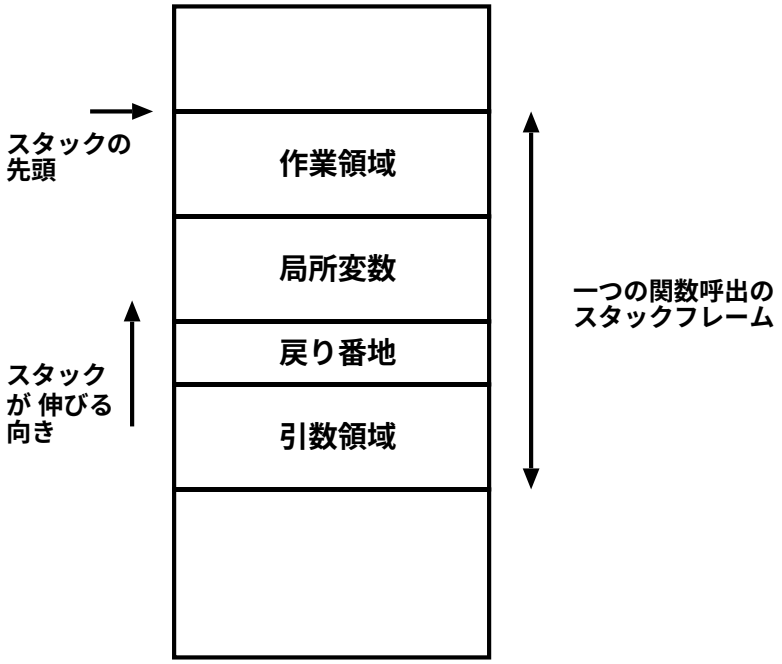

図 3.1: スタックフレーム

関数型言語の変数の参照は、トップレベルの変数の宣言だけでなく、関数の中で宣言された局所的な変数にもアクセスする必要がある。従って、プログラムではフレームポインタを処理する必要がある。すなわち、参照された変数を含むフレームを検索するためにこれらのフレームを飛ばさなければならない。従って、フレームはスコープの見え方を表す連鎖であるもう１つのリンクで結ばれている。この連鎖は [**静的リンク**] と呼ぶ。

パイプライン上の問題点

関数型言語が通常のパイプラインでは、効率は良く実行できないのは、プログラムの中で頻繁に起きる関数の呼び出しによる、スタック上のフレーム操作の命令の存在である。スタックの操作は以下の場合に通常のパイプラインではハザードを生じる。

例えば、次の関数 f の評価は、その中で関数 g の呼び出しがあり、さらに g の中では関数 h の呼び出しが行われている。$f \rightarrow g \rightarrow h$ 順に呼び出しが行われている。通常のパイラインの場合は、図3.2 ように１つのスタックの上でフレーム操作が生じ、パイプラインのストールが起きる。フレーム操作の命令列は、レジスタの設定とフレーム内のメモリアクセス操作からなり、レジスタ設定とメモリアクセス操作との間では、パイプラインストールが生じる。またメモリアクセスはキャッシュミスが生じればストールとなる。

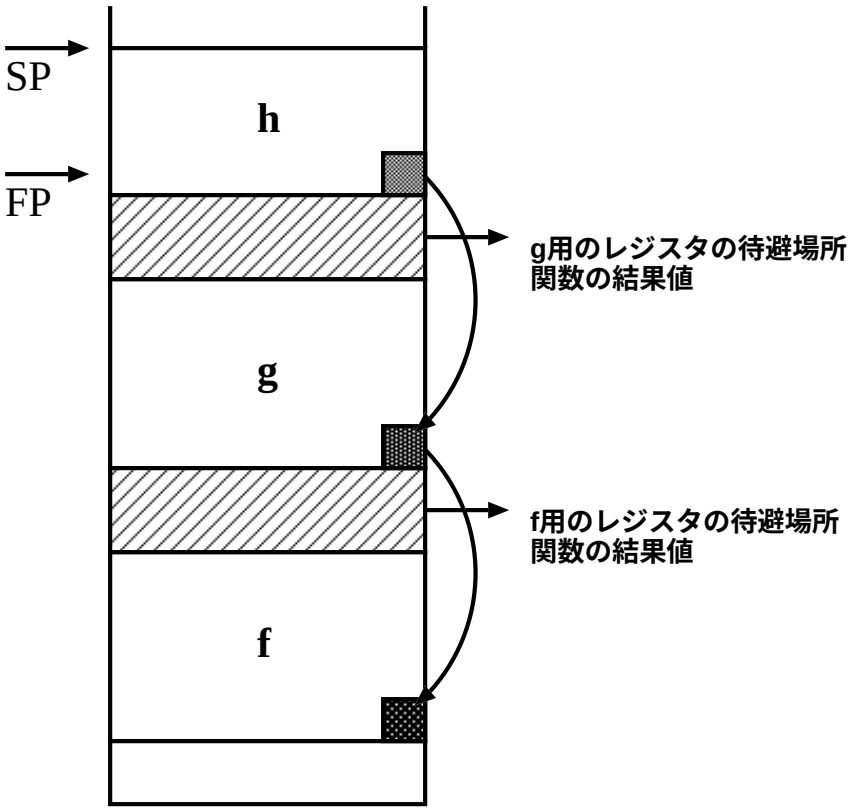

図 3.2: 通常パイプラインでのフレーム操作

しかし、マルチスレッド型プロセッサの場合は、図 3.3 それぞれの関数 f、g、h を異なるスレッド Th1、Th2、Th3 に割り当て、Th1 は Th2 の結果が出るまで、サスペンドし、その間は別のスレッドに実行を割り当てることにより、プロセッサを休まずに走らせることができる。その場合も、各スレッドはフレームを持つ必要がある。このフレームをスレッドフレームと呼ぶことにする。フレームが操作が起きるが、各関数は別々のフレーム（並列アクセス可能な）を持ち、スレッド毎にレジスタを保持することにより、パイプラインでもフレーム操作のストールを解消できる。

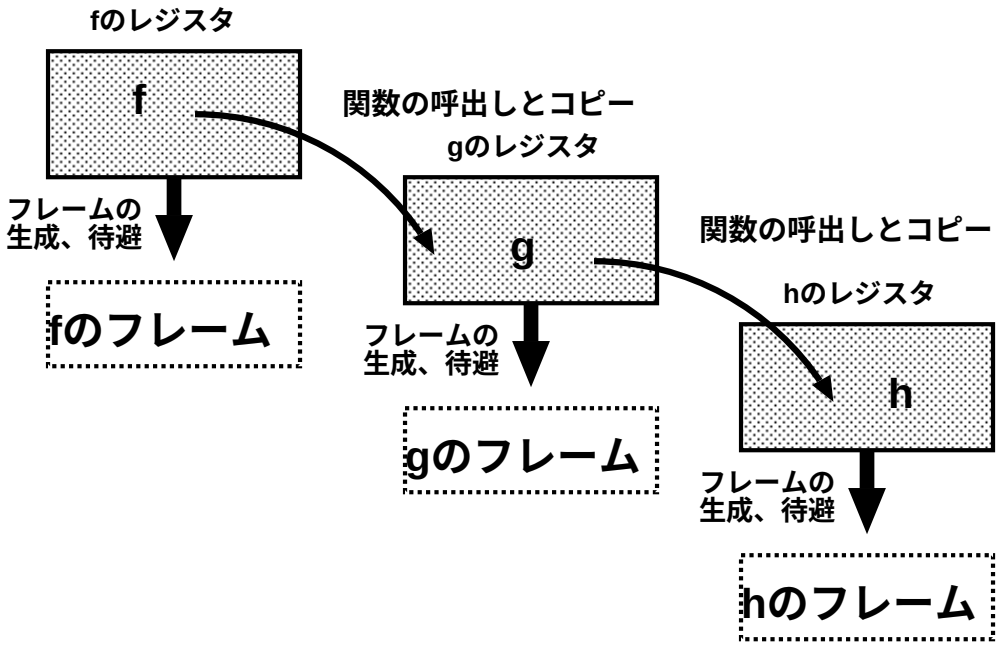

図 3.3: マルチスレッド型プロセッサでのフレーム操作

3.2 Standard ML の並列実行性

3.2.1 関数型プログラムの並列処理機構

関数型プログラムにおける並列処理構造は、並列実行とパイプライン実行に分かれる。

1. 並列実行 - 互いにデータ依存性を持たない独立な関数の間で可能である。例えば、
   - 関数引数の並列評価 ： お互いにデータ依存がないので、並列に評価できる。
   - リストやアレイ要素の並列評価
2. パイプライン実行 - 互いにデータ依存関係を持つ関数の間で、ストリームやリストのようなデータ列をパイプライン処理する。例えば、
   - 関数適用側と関数本体の間のパイプライン処理
   - リスト (ストリーム) あるいはアレイの生成と使用の間のパイプライン処理 ： すべての処理が完成するのを待たずに、一部のデータを先に使用側に渡す時は、生成側と使用側の実行は並行に行うこたができる。図 3.4 は、生成側の関数 producter が平方数列を作り、使用側の consumer がこれらを順次に使って和を取る並行なストリーム処理である [4]。

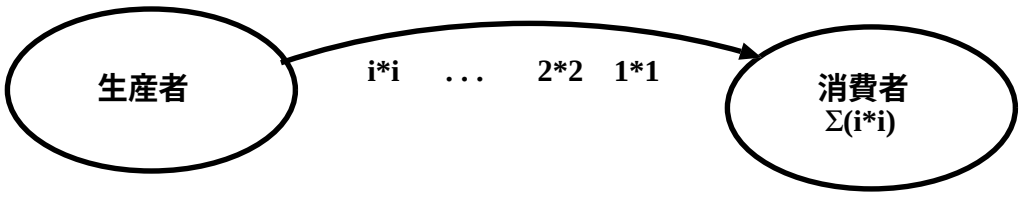

図 3.4: ストリム処理

3.2.2 SML 並列実行可能性

SML の通常の処理系ではプログラムに内在する並列実行可能性が無視されている。Standard ML の核言語の構文構造の中心は**式**である。従って並列に実行が可能な式を見い出せば、プログラムの並列検出ができたことになる。**式**は、その最小単位である**原子式**、原子式が関数適用で結び付いた**適用式**、さらに適用式を中置演算子で結び付けた**中置式**を通じて、階層的に定義できる。表 3.1 は、式の文法である。この文法から並列実行可能性を見つけることができる。例えば、

- 中置式の場合、exp_1 op exp_2

この時、逐次に評価を行うのではなく、両式 (exp_1,exp_2) を並行に評価を行い、両式の同期を取って、次の段階の評価を行う。

appexp	::=	atexp	
	|	appexp atexp	関数適用 (左結合)
infix	::=	appexp	
	|	infix op infix	演算子式
exp	::=	infix	
	...		(–中略–)
	|	exp **andalso** exp	論理式
	|	exp **orelse** exp	論理和
	|	exp **handle** match	例外ハンドラ付き式
	|	**raise** exp	例外発生式
	|	**if** exp **then** exp	条件式
		else exp	
	|	**while** exp **do** exp	繰り返し式
	|	**case** exp **of** match	場合分け構文
	|	**fn** exp	関数式
match	::=	$mrule\langle \mid match\rangle$	パターンマッチング
mrule	::=	pat⇒exp	

表 3.1: SML 式の文法 [5] [6]

• 値を変数に束縛する値の宣言文が複数並んでいる場合、

val $x_1 = exp_1$; ... ;val $x_n = exp_n$;

通常の処理系では、この場合は順番に評価を行うので、効率はあまり良くない。各式の間に依存関係がない場合は簡単に並列に実行することができる。もし、依存関係があったとしても、ただ依存関係のある部分だけを待ち状態して、並列できる所はできる限り並列に実行を行い、依存関係が解決するとただち評価を開始すればよい。

• 複数の局所宣言文の場合、

let dec **in** exp_1; ... ;exp_n **end**

式の間で並列実行も考えられる。

• 条件式の場合、**if** exp_1 **then** exp_2 **else** exp_3

通常の ML の処理系では、無駄な処理を避けるために条件式は、述語 exp_1 の評価を行った後で exp_2 か exp_3 を評価を行う。述語 exp_1 の評価が終わるまでは、次の評価ができない。しかし、先に、両式を並列に評価して置けば、述語の評価が終り次第にすぐ値を得ることができる。

以上の議論から分かるように、並列検出のコンパイラを実装した関数型言語の実行はマルチスレッド型プロセッサの導入で高効率を得ることができる。

第4章　コンパイラの実装

4.1　目的と内容

第3章では、Standard ML の特徴、文法、処理方式について確認をした。本章では、言語の文法と性質に着目しながら、後半の評価を行うためにコンパイラの実装を行う。実装はコンパイラの全過程ではなく、字句解析、構文解析、記号表管理などの三つの部分を実装を行う。

4.2　字句解析

この節では、字句解析の仕様を規定するための技法と、実現法上の注意点を述べた後、実装した方法を紹介する。

字句解析については、正規表現によって、字句を定義すれば、そこから字句解析を行うプログラムを生成することができる。すなわち、字句の仕様をパターンとして与えるだけで、字句を取り出すたののプログラムが自動的に生成できるようになった。Unix の **Lex** がそのツールである。

しかしながら、本研究では、そういうツールを使わず、すべてを SML を使って実装している。

4.2.1　字句解析の役割と基本概念

字句解析の役割

字句解析はコンパイラの最初のフェーズである。その重な仕事は、文字を順に読み込みながら、構文解析の要求に応じてトークンの列を出力する。この関係を図 4.1 に示す。字句解析はほとんど構文解析ルーチン中のサーブルーチンまたはコールルーチンで実現するのが普通である。字句解析は、” 次のトークンの獲得” という要求を構文解析から受け取ると、入力文字を読み込んでいって、次のトークンを確定する。

コンパイラの解析フェーズを字句解析と構文解析に分ける理由は次の通りである。

1. 最大の理由は、解析フェーズを２つに分けると、設計が簡単になるからである。
2. コンパイラの効率が向上する。字句解析はその仕事だけに専念すればよく、そのため

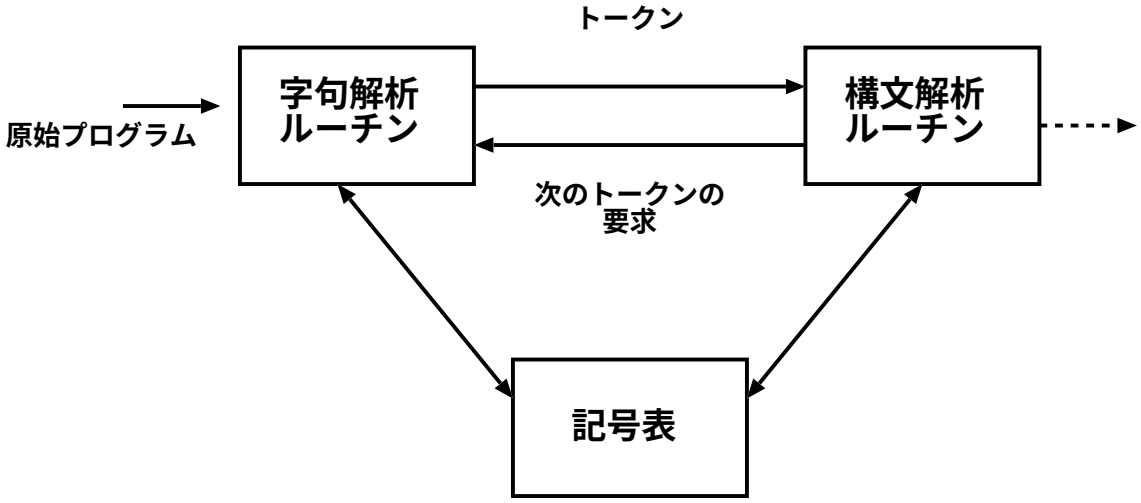

図 4.1: 字句解析と構文解析の関係

に効率のよい処理ができる。コンパイラの時間の大半は、ソースコードを読み込んで、トークン列を作り出す仕事に費やされるので、それらの処理のために特別なバッファリング技法を用いると、コンパイラの性能は大幅に改良される。

3. コンパイラの移植性が高かまる。

その基本概念

これからの字句解析の説明では、“トークン”、“字句” および “字句値” という用語をよく使うので、それらの違いを使い分ける必要がある。表 4.1 にこれらの用語の例を示す。一般に、入力に現れる文字列の中には、字面上の表現が異なっても、トークンとしては同じものである。ソースコードの中に現れる文字列で、トークンのパターンと合致するものを、そのトークンの字句という [8]。

トークン	字句の例	パターンの説明
val	val	val
if	if	if
relation	$<, <=, =, <>, >, >=$	< または <= または = または <> または > または >=
ID	$x, y, power, cube$	英字で始まる英数字の列
NUM	$100, 23.43, 3.14$	任意の数値定数
literal	“$name$”	“と” の間の文字列

表 4.1: トークン、字句、パターンの例

トークンの属性

パターンが複数の字句と合致する場合は、後段のフェーズでの処理のために、字句解析は字句に関する情報も後のフェーズに渡さなければならない。たとえば、NUM は 10 とも、12 とも合致するが、コード生成にとっては、NUM が実際にどの数字に合致したのが本質的に必要な情報になる。

そのために、トークンには属性を付加する。実用上はトークンの属性を記号表エントリへのポインタだけとし、トークンに関する情報は記号表の中に入れておくのが普通である。

4.2.2 トークンの規定と認識

トークンの規定

正規表現はパターンを規定するための重要な記法である。各パターンは文字列の集合と合致するので、正規表現はそれらの集合に対する一種の名前としての役割を果たす。

正規表現を使って、識別子と数字列を表 4.2 のように表現することができる。

letter	$\longrightarrow$	$A \mid B \mid C \cdots \mid Z \mid a \mid b \mid \cdots \mid z$
digit	$\longrightarrow$	$0 \mid 1 \mid \cdots \mid 9$
ID	$\longrightarrow$	**letter** (**letter** $\mid$ **digit**)*
digits	$\longrightarrow$	**digit digit***
optional-fraction	$\longrightarrow$	**.digits** $\mid \varepsilon$
optional-exponet	$\longrightarrow$	$(E(+ \mid - \mid \varepsilon)$**digits**$) \mid \varepsilon$
num	$\longrightarrow$	**digits optional-fraction optional-exponent**

表 4.2: 正規表現による数字列と識別子の表現

トークンの認識

節 4.2.1 では、トークンをどのように規定するかを述べた。本節では、トークンをどのように認識するかを考える。

字句解析を作成する時の途中の段階として、まず、遷移図または状態遷移図と呼ぶ一種の流れ図を作る。遷移図には、開始状態というラベルを持つ状態が 1 つある。この状態は、トークンの認識を始める時に、決って制御が与えられる初期状態を表す。状態には、そこに制御が移った時点で実行される動作を付けてもよい。ある状態に制御が移ると、そこで次の入力文字を読み込む。そして、現在の状態から出ている辺の中で、入力文字と一致するラベルの辺があれば、次は、その辺が指している状態に遷移する。その辺がなけれ

ば、照合は失敗である。もし失敗が起きると、前進ポインタを 1 文字分もとに戻しておかなければならない。

一般に、遷移図は１つだけでなく、いくつがあってもよく、各図ごとに一定のトークンの集まりを規定することができる。１つの図を辿っているところに失敗が起ったら、前進ポインタをその図の開始状態の位置に戻し、次の遷移図を起動すればいい。

トークンを認識するのにもう１つ重要なのは、トークンに対する字句はパターンと合致する字句のうちで最長な記号列を選ぶべきである。例えば、12.3E4 入力があったとすると、字句解析は 12 や 12.3 までで処理をやめてはいけない。最長の記号列を選ぶべきである。[**最長一致原理 longest match principle**]

4.2.3 字句解析の実装

(1) トークンの規定

字句解析を実装を行う前に、トークンを規定する必要がある。トークンを規定のするために、型といくつかの約束をする。まず、基本データ基本型は **int** にし、その上でのほかの複合型を決める。例えば、**int** を要素とするリストと組を考える。また、キーワードは識別子として使うのを禁じる。その上でのキーワード、識別子、数字列、特殊記号および制御記号がトークンを構成する。予約語、識別子、数字列とトークンを次の表 4.3 のように正規表現を用いて規定する。

トークン	::=	予約語	識別子	数字列	特殊記号		
予約語	::=	andalso	orelse	if	then	else	fun
	|	val	let	in	end	ture	false
識別子	::=	英字	識別子	英数字			
英数字	::=	英字	数字				
数字列	::=	数字	数字列	数字			
英字	::=	A	B	···	a	···	z
数字	::=	1	2	3	···	9	0

表 4.3: トークンの規定

(2) トークン分類のためのデータ型定義

プログラムの最初として、前述のトークンを分類するデータ型を定義する必要がある。これはユーザ定義型 (**abtatype**) を用いて定議する。予約語、トークンの定義は次のようになる。

```
datatype keyword = ANDALSO | ORELSE | IF | THEN | ELSE | FUN
| VAL | LET  | IN | END  | TRUE | FALSE

datatype token =
  EOF | Reserved of keyword
| ID of string | NUM of int
| BANG (* ! *) | DOUBLEQUOTE (* " *)
(* 省略 *)
```

この token 型は、keyword 型の値を持った Reserved か、 string 型の値を持った ID か、または int 型の値を持った NUM かのどれかの値を持つことを示す。

(3) 字句解析処理

ソースコードを上のトークン型に分解する字句解析処理は、以下の処理の組み合わせとして実現する。

1. 空白の読み飛ばし処理
   通常、プログラム言語では、字句と字句の間に任意の空白（スペース）や改行を入れてよいというものが多い。そのために、字句解析の最初の仕事は、これらの空白文字を読み飛ばし処理が必要である。
2. トークン種類の判定
   空白を読み飛ばした後、空白以外の最初の 1 文字を先読み出し、これから読み込み語彙が予約語、識別子、文字列、特殊文字、制御文字のいずれかを判定する。
3. 常に 1 文字先読みをする
   この実装では、常に 1 文字を先読みをするという立場でプログラムを書くことにする。なお、より複雑な字句を扱う場合は一般に複数文字の先読みが必要になる。その場合は 1 文字だけでなく複数文字を戻すことが必要になる。また、次の構文解析の実装でも、常に 1 字句 (トークン) の先読みで構文解析を行う。
4. トークンの読み込み
   判定した語彙の種類に応じて実際の読み込み処理を行う。

この四つの方針に基づいて、字句解析関数を実現する。それぞれの四つの処理も関数として実装し、字句解析関数の中でその四つの関数を呼び出すの形で全般を構成する。3 番目の処理は、ライブラリ関数 **lookahead** (fn : TextIO.instream → TextIO.elem option) で実現した。

実装した字句解析関数によって、次のようなトークンを返すことができる。

```
val x = 9;
Reserved(VAL)
```

```
ID(x)
EQUALSYM
NUM(9)
SEMICOLON
```

4.3 構文解析

どんなプログラム言語でも、プログラムが構文的に正しい形をもつために、構文構造を規定する規則がある。この構文を規定する規則を文法あるいは構文規則という。構文的に正しい形を持つプログラムのみが処理可能となる。例えば、SML では、プログラムは式の集まりであり、式はトークンから、というように構成される。プログラム言語の構成を定める構文は、文脈自由文法や BNF(Backus-Naur) 記法を用いて記述できる。この節では、BNF 記号を用いて文法を記述する。文法は次のいくつかの点でコンパイラの作成にとって重要である。

- 文法は、プログラム言語の構文に関する仕様を正確でしかも理解しやすい形で示す。
- ある文法からは、原始プログラムが構文的に正しい形をしているかどうかを効率よく判断できるし、構文上での曖昧さも検出できる。
- 文法を正しく設計すれば、文法からプログラム言語の構造が明確となり、その構造は原始プログラムを目的プログラムに翻訳したり、誤りを検出したりするのに役立つ。

この節では、最初に基本的な概念を説明し、その後に構文解析に用いた手法として下向き型構文解析を紹介する。最後に、実装のことを説明する。

4.3.1 構文解析の役割と基本概念

構文解析は、字句解析から受渡されたトークンの組合わせを、文法の意味にマッチさせ、結果として木構造に変化する処理である。もし、構文誤りがあれば、使用者に分かりやすい形で誤りを知らせ、回復処理をして残りの入力の処理を続ける。分かり易く言えば、構文はプログラム中のトークン列の並べ方の規則である。

通常は、構文は文脈自由言語で定義し、BNF(Backus-NaurForm) で表現する。例えば、簡単な式の構文の定義 (BNF 表記による) は次のようになる。

式 ::=		式は
	式 '+' 式	式とプラスと式からなる
|	式 '*' 式	または式とスターと式からなる
|	'(' 式 ')'	または式に括弧を付けたもの
|	ID	または識別子である
|	NUM	または整数定数である
|	;	または空である

表 4.4: BNF による式の表現

構文解析はトークン列から、文法に従って構文木を返す。例えば、表 4.4 の文法による式 (5 + x) * y は構文解析を通じて、次のような構文木を生成する。

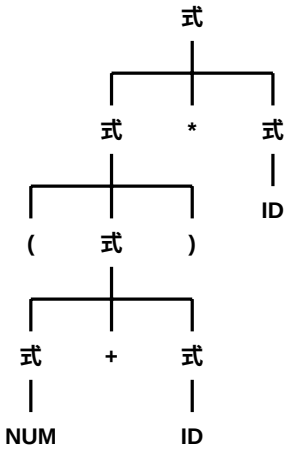

図 4.2: 簡単な式の構文木

4.3.2 構文解析法

構文解析法は数多くあるが、大きく下向型構文解析と上昇型構文解析に分けられる。解析木を根（文法の開始記号）から葉（字句）の方向に作って行くのが下向型構文解析（top-down parsing）であり、逆に葉から根の方向に作って行くのが上昇型構文解析（bottom-up parsing）である。

下向構文解析は、文法規則の元で、開始記号から木を下向きに作っていく。この解析は最左導出を順に作っていくこととである。解析途中の段階の状況は図 4.3 左のようになる。それと逆に、与えられた文法規則の元で、入力記号列から、葉から根つまり文法の開始記号の方向に作って行くの上昇型構文解析は、最右導出を逆順に作っていく。解析途中の状況は図 4.3 右のようになる。上昇型構文解析の基本操作はシフト（shift）と還元（reduce）である。

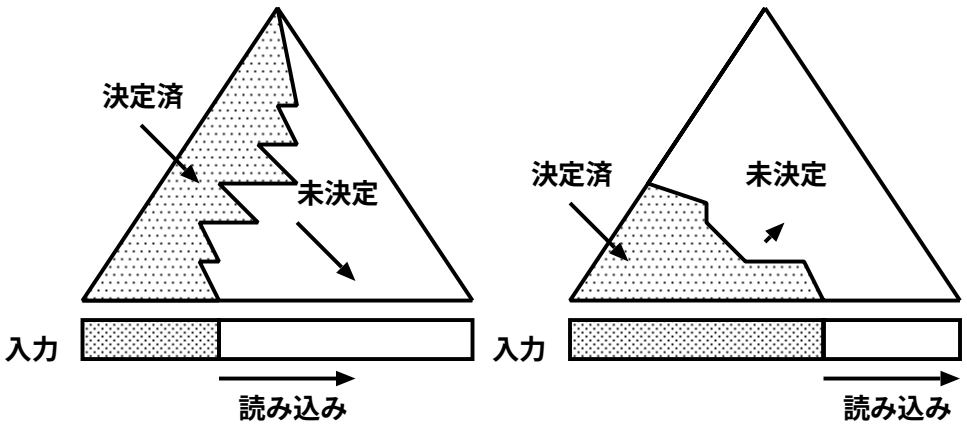

図 4.3: 下向型と上昇型構文解析による解析状況

4.3.3 構文解析の実装

本研究では、構文解析を下向き型構文解析の再帰下降解析を用いて実装する。再帰的下向型構文解析は文法の非終端記号毎に、解析関数を作ることであり、文法と構文解析プログラムの対応が図 4.4 のように分かりやすい。

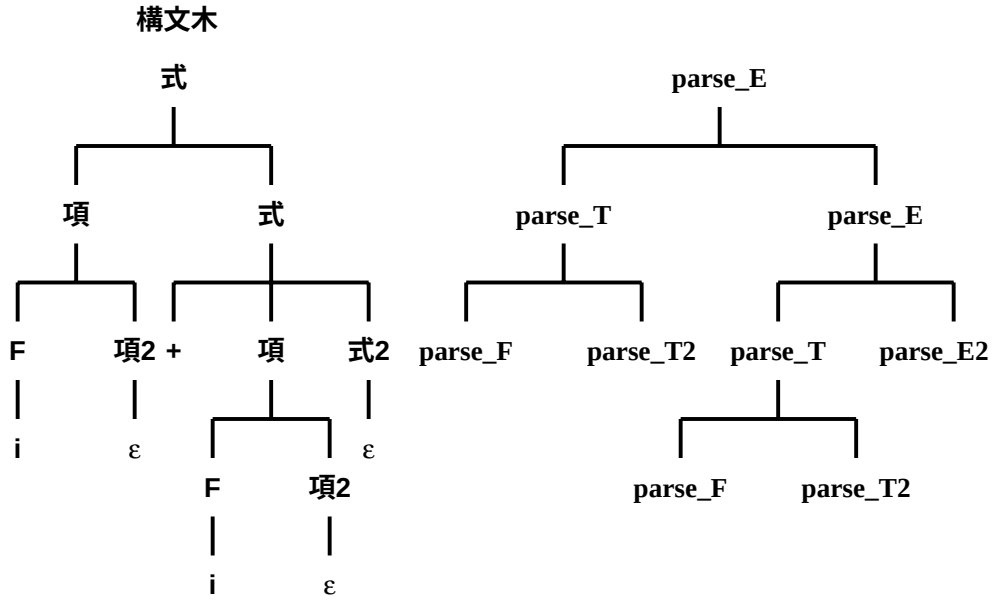

図 4.4: 式 i+i の構文木と解析関数の対応

再帰的下向型構文解析は文脈自由文法のある部分に対してのみ適用できる。どの文脈自由文法に対してもうまく働くわけではない。しかし場合によって、構文規則を改良することにより効率良く、行くようにできる。

再帰的下向型構文解析の効率が悪くなる場合が、２つがある。

1. 左再帰の存在である。
   例えば、次のような規則があったとする。

   $$\mathbf{A} ::= \mathbf{A}a \mid b$$

   $\mathbf{A}$ に対応する解析関数の本体では、最初にいきなり自分自身を呼ぶことになる。これにより、実行は無限ループに陥り、解析は止まらない。この時、$\mathbf{A} ::= b \mid \mathbf{A}a$ のよう終端記号を先に適用すれば、効率は悪いが解析は可能である。
2. ある非終端記号は二通り以上の可能性がある場合である。

   $$\mathbf{A} ::= s \mid t$$

   この時、入力記号列の現在だけの情報だけでは、うまく行かない。そのとき、入力記号列をさらにいくつか先読みする必要がある。

左再帰が存在する場合は、元の文法を同じ言語を生成する文法に変更し、左再帰を除去する。例えば、1 番目の場合は次のように同じ言語を生成する文法に変換することができる。

$$\mathbf{A} ::= b\ A_1$$
$$A_1 ::= \varepsilon \mid aA_1$$

2 番目の場合、左括り出し (left factoring) という手法を使ってうまく行くようにできる。例えば、$\mathbf{A} ::= s \mid t$ に対して s と t がそれぞれ同じの先頭 u を持っている場合、次のように変換ができる。その v と w が同じ先頭を持ってないと仮定する。

$$\mathbf{A} ::= u\ A_2$$
$$A_2 ::= v \mid w$$

上に述べたことを注意しながら、文法を定める。また、SML 言語のプログラム構文規則全部ではなく、サーブセットを取って実装を行う。実装に使った文法は次のようになる。

文法 4.5 に従って、構文解析関数を作成する。文法 4.5 に現れる非終端記号に対して、それぞれの解析関数を作る。フレーズ、宣言文、式、式 4、式 3、式 2、式 1、式 0、基底式各々に対して、解析関数を別々に宣言する。上の文法による構文解析処理の中で、曖昧なく構文できるように

(先読みトークン、構文木)

の組を受け渡しながら処理を続ける。

構文木は次のようにユーザデータ型により定義する。

```
datatype program = Phrase_list of definition list
and definition = Dec of dec  | Exp of exp フレーズ
and exp = VarExp of var |  IntExp of int (*省略する*) 式
and dec = FunctionDec of fundec | ValueDec of valdec 宣言文
withtype fundec = {name: symbol, params: symbol list,body: exp}
```

実装した構文解析操作の例:

```
- run(); <<<--- は構文解析のスタート。
%%% fun f (x,y) = x + y;
Dec(FunctionDec{name=f,params=[x,y],body=OpExp{left=VarExp
(SimpleVar x),oper=PlusOp,right=VarExp (SimpleVar y)}})
%%% let val x = 2 in 9 * x end;
Exp(LetExp{decs=ValueDec{name=x,init=IntExp2},
body=OpExp{left=IntExp 9,oper=TimeOp,right=VarExp (SimpleVar x)}})
%%% val y = 188;
```

プログラム	::=	phrase_list					
phrase_list	::=	フレーズ					
	|	フレーズ	phrase_list				
フレーズ	::=	宣言文					
	|	式					
式	::=	式 4					
	|	LET	宣言文	IN	式	THEN	
	|	IF	式	THEN	式	ELSE	式
式 4	::=	式 3					
	|	式 3	,	式 4			
式 3	::=	式 2					
	|	式 2	=	式 3			
	|	式 2	<>	式 3			
式 2	::=	式 1					
	|	式 1	+	式 2			
	|	式 1	-	式 2			
式 1	::=	式 0					
	|	式 0	*	式 1			
	|	式 0	/	式 1			
式 0	::=	基底式					
	|	基底式	式 0				
基底式	::=	整数					
	|	識別子					
	|	論理値					
	|	(	式	)			
	|	()					
宣言文	::=	VAL	式				
	|	FUN	式				

表 4.5: 文法の規定

```
Dec(ValueDec{name=y,init=IntExp 188})
%%% 1 + y;
Exp(OpExp{left=IntExp 1,oper=PlusOp,right=VarExp (SimpleVar y)})
%%%
```

例の中で%%%はプログラムの促進プロトタイプであり、局所宣言文 **let val** x = 2 **in** 9 * x **end**; と 1 + y; の入力より抽象構文木

```
Exp(LetExp{decs=ValueDec{name=x,init=IntExp
2},body=OpExp{left=IntExp 9,oper=TimeOp,right=VarExp (SimpleVarx)}})
```

と

```
Exp(OpExp{left=IntExp 1,oper=PlusOp,right=VarExp (SimpleVar y)})
```

が得られ、それぞれは図 4.5 の (1) と (2) のような構文木に対応できる。

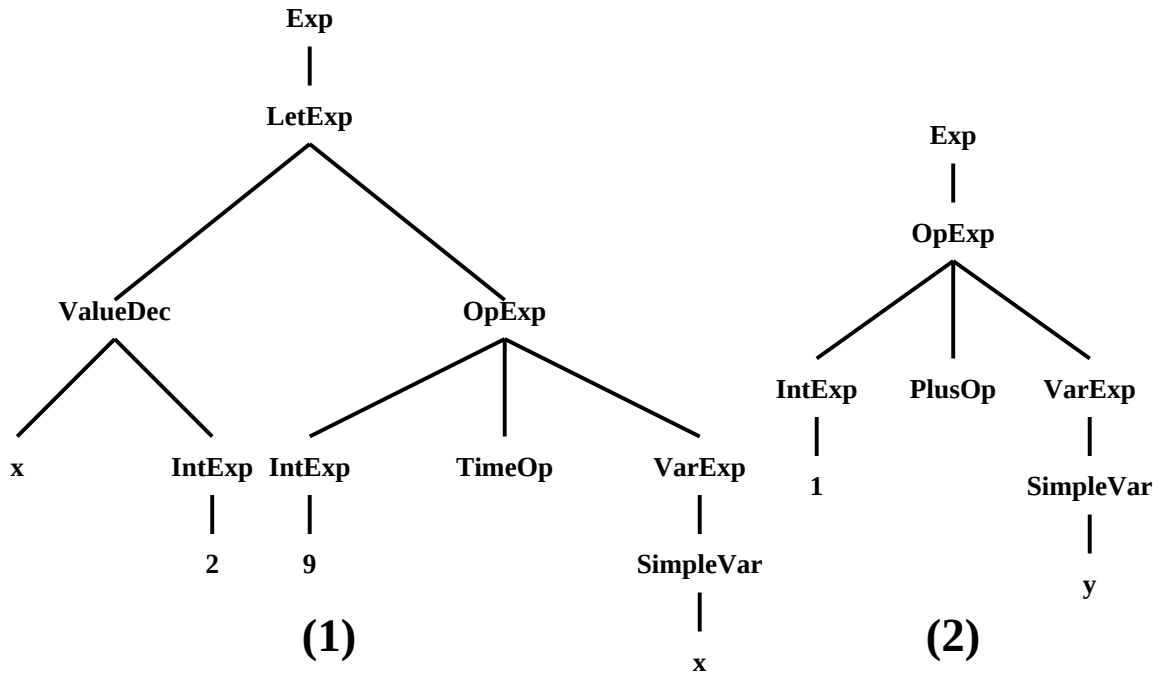

図 4.5: 式に対応する構文木

4.4 記号管理表の作成

記号表の役割は、ソースコードに含まれる情報のうちで直接中間コードや構文木に反映されない部分（名前や型などの情報）を蓄えることである。一方で、多くのコンパイラでは名前のスコープなどの処理も記号表の機能に含めて実現される。本節では記号表の基本概念とそれらの実現方法を述べ、次に識別子の有効範囲の管理を議論し、最後に本研究で実装した記号表の手法を述べる。

4.4.1 基本概念と実現方法

記号表の内容

変数名、関数名などは、ソースコードにおいて何らかの情報を持っている。これらは、プログラムの宣言部において、型、値などの情報が宣言される。コンパイラは、宣言された変数名、関数名の情報を記号表（symbol table）に保存し、次のフェーズで必要によってその内容を参照する。プログラミング言語において、識別子の種別によって、それらが持っている情報も違って来る。共通なのは、識別子の名前と種別の区分である。

異なる種別、変数名と関数名の情報は次のようになる。

- **変数名**
  - 型
  - 大きさ
  - スコープ範囲
  - その変数名が宣言されているか、いないかの情報
  - プログラムの実行時に割り当てられる番地
- **関数名**
  - 仮引数の個数、およびそれらの型
  - 結果値の型
  - 有効範囲に関する情報
  - 関数の先頭番地

記号表に対する基本操作

1. **検索**
   ある名前が記号表に入っているかどうかを探す操作である。検索は、名前が宣言される時にも、その名前を使用する時にも行う。そのために、検索の効率が非常に重要になっている。

2. **登録**
   記号表に新しいエントリを書き込む操作である。

探索の方法にいろいろがあって、高速にできるハッシュ法と 2 分探索があり、実現が簡単な線形探索もある。
記号表の実装も、ハッシュ法、2 分木、線形リストとして実現できる。

4.4.2 識別子の有効範囲の管理

記号表エントリは、名前の宣言に関する情報を保持しておくためのものである。ソースコードに現れる名前について、記号表を探索する時には正しい宣言を持つエントリを返さなければならない。名前の宣言がいくつかある時に、どの宣言が適用されるかは、ソース言語の有効範囲規則によって規定される。
ブロック型スコープ規則 [23] では、プログラムは入れ子状になった複数のブロックからなり、あるブロックでの名前の定義はそのブロックの内部全体を有効範囲とするが、もしそのブロックの中に、別のブロックの同じ名前の定義が含むとその元々の宣言は無効になる。
ブロック型スコープ規則を記号表の観点から見ると、プログラム上である名前が出現した時、その場所にちょうどその名前の定義があると、それを参照するだけで済む。だがもし同じ名前に対して、複数の定義があるとそれを囲む一番内側の名前を参照すればいい。ブロック型言語の場合は、識別子の有効範囲の管理は次のように実現できる。

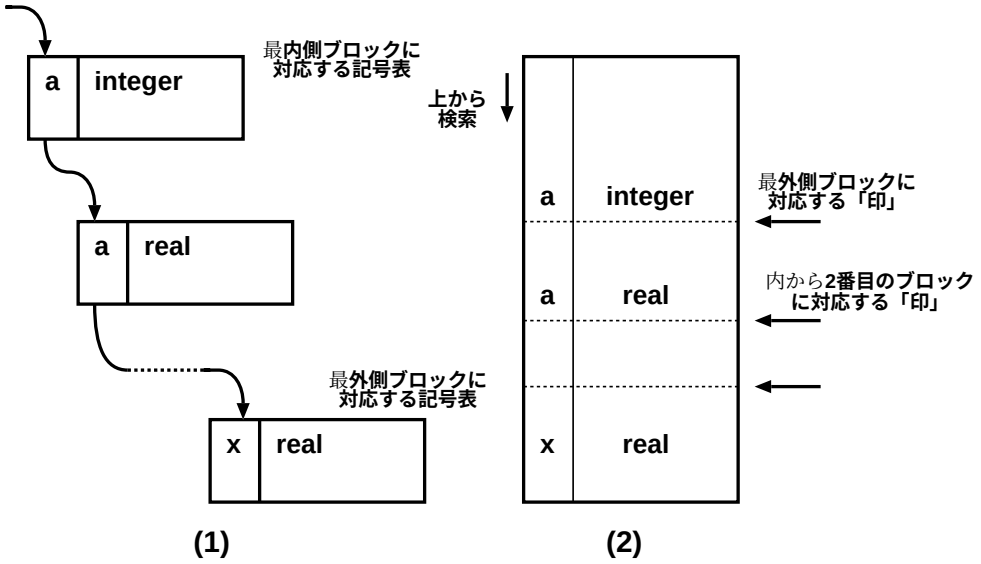

図 4.6: ブロック型スコープ規則の実現

- 各ブロックごとに別の記号表を用意する。図 4.6 の 1 のように探索において最内側のブロックから順に外側のブロックへ探索を行う。
- １つの表で済む場合もブロック型スコープ規則の「一番新しいスコープは一番最初に閉じられる」という性質を利用して実現する。例えば記号表が単純な 1 次配列で実現されている場合、図 4.6 の 2 のようにブロックに入るごとに現在表のどこまで定義が入っているかの「印」をつけ、定義の登録は常に表の最後に追加する。検索においては表の末尾から前に向かって探索し、最初に見つかったものが最内側の定義に対応する。

4.4.3 記号表の実装

設計方針

設計方針としては、まず言語の特徴をできるだけ生かし、しかも言語機能上およびコンパイラ技法上の本質的なものはできるだけ残したい。最後にコンパイラ · プログラムを小さくしたい。その三つの方面を考えながら記号表の設計を行う。

1. 構文木と記号表を合わせたのが、意味的にソースコードと合致できるように、構文木に現られない情報をすべて登録すること。
2. 記号表を変数名用と関数名用を別々にして実装する。
3. 記号表のアクセス効率が重要であるが、本研究では記号表の実装を単純な配列として実装する。
4. 構文文法を定める時、データ型を **int** だけにしたので、記号表で型の情報持たないように設計する。
5. ある識別子を登録する際に、その識別子が値が決っていると、その値も記号表に登録を行う。
6. 3.1.1 節に述べたように、SML の変数は静的スコープ規則に従っているので、その方式も記号表の中で実現する。
7. 関数名の記号表では、引数の型検査を実装しないが、引数の数の検査は行うように実装する。
8. 変数の参照するのに前議論したブロック構造に類似の方法で実装する。
9. SML の特徴である同じ変数に対して、何度でも宣言できることによる、変数の参照 (参照透明性) の問題も実現する。

変数名用記号表の実装

変数名用の記号表に関連する宣言を集めると次のようになる。

```
type binding = {count:int, level:int, value:{flag:int,boundvalue:int} list}
type bucket = (string*binding) list
type table = bucket Array.array
val t : table = Array.array(SIZE,nil)
```

記号表を配列として実装している。配列の要素は**bucket**、また**bucket**は**string*binding**型を要素とするリストで実装する。**string**型は識別子の型である。**binding**は**int**型のラベル**count**とラベル**level**、レコードを要素としているリストのラベル**value**で構成されている。

- ラベル**value**の要素**flag**は値が決っているか否かの判定に用いる。もし、**flag**が1にセットされているとその識別子の値は決っていることを示す。この値は**boundvalue**を参照することで得られる。もし、0にセットされている場合は、値の未決定のことを示し、その変数への参照できない。
- **count**は同じ名前の値を区別するために使用する。
- **level**は同じ名前の値が局所的な宣言で出現する時、それらを区別するために導入した。この実装では、値の宣言が局所的に行われる場合、**count**は変化しないので、**level**を導入により参照の正しさを保証する。局所的な宣言が**let**文の入れ子構造で行われると、それに応じて**level**の値も増加または減少される（内側のブロックを出る時)。

実装の中で、局所的な宣言による変数の記号表への登録関数とグロバールな宣言による変数の登録関数を別々に設計している。

後は、局所的な文を出る度に（内側ブロックから出る)、leave関数を起動して**level**の値を減少する。最後の局所文を出ると、**level**はグロバール時の状況の値になる。それぞれの実装した記号表は図4.7と図4.8になる。

関数名用記号表の実装

関数名の記号表に関する宣言は次のようになる。

```
structure A = Ast
type f_binding = {f_count:int, f_level:int,
value:{parameter:parame list, body:A.exp} list }
type f_bucket = (string*f_binding) list
type f_table = f_bucket Array.array
val f_t : f_table = Array.array(FUNCMAX,nil)
```

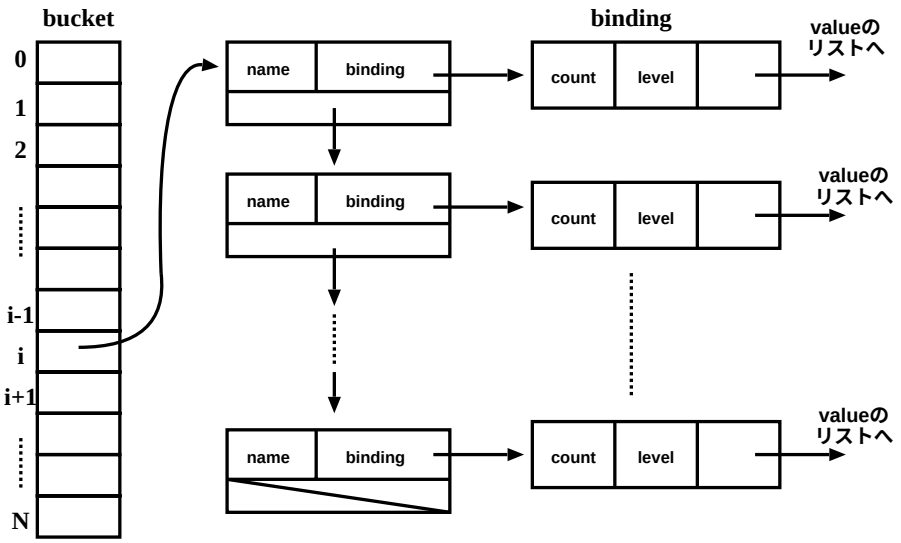

図 4.7: 記号表の様子

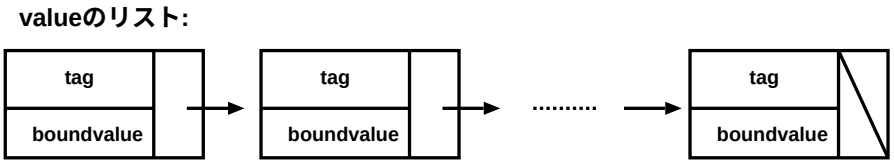

図 4.8: value リストの要素

関数名の記号表の実装は変数名の場合と似ている。**f_count** と **f_level** は変数名の場合と同じ働きをする。違いは **value** の中身である。この中には関数の引数と関数定義の本体が入っている。複数の引数のためにリストとして実現した。本体は式で宣言している。**value** は図 4.9 のように表現している。

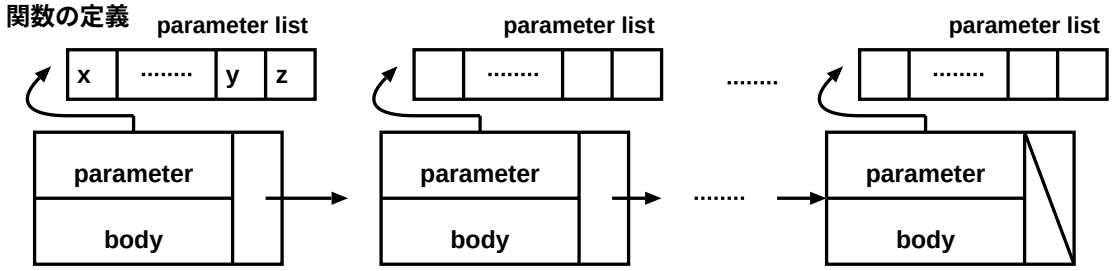

図 4.9: 関数記号表の value リスト

第5章　アーキテクチャの設計

5.1　目的

これまでの章で、関数型言語の特徴と問題点やマルチスレッド型プロセッサの特徴を確認し、コンパイラのフロントエンドの実装を終えた。本章では、提案する手法の評価を行うため、マルチスレッド型プロセッサアーキテクチャを設計する。また、比較評価の対象として、通常パイプラインプロセッサの設計も与える。

5.2　マルチスレッド型プロセッサアーキテクチャ

実際、伊藤らにより関数型プロセッサの実行に適したマルチスレッド型アーキテクチャが提案されている [17]。このプロセッサはスレッド数と同数のレジスタセットを持ち、次に実行されるスレッドを選択するスレッド選択ユニットを持っている。これにより、データハザード、分岐ハザード、構造ハザードは避けられている。また、関数型言語が十分なスレッドが生成できるという前提として、パイプラインステージを 17 段のスーパパイプラインとして設計し、スレッド毎にプログラムカウンタ、レジスタファイル、および各種レジスタを有する [1] [16]。そのハードウェアの構成は以下の図 5.1 のようになる。

このプロセッサは四つの特徴がある。

1. このプロセッサはステージ数分のスレッドで満たされるようにするための十分なスレッドが必要である。一度スーパパイプラインのステージが満たされれば、このプロセッサはピーク時の性能で実行し続け、追加的にスレッドを加えても高速化できない。
2. このプロセッサは単一のプロセッサ環境で多くのスレッドをサポートする。
3. このプロセッサはパイプラインステージと同数のスレッドを発行することによりパイプラインストールを隠蔽する。
4. このプロセッサは複数スレッドからの命令を１つのパイプラインを共有して実行する。その時、スレッドを識別する必要があり、スレッド **ID** を導入し、問題を解決した。

一方で、このプロセッサは 17 段の命令パイプラインを有しているが、関数型プログラムを実行させることによる実際の効果を証明することには至らなかった。しかし、マルチスレッド処理を導入するとパイプライン · ストールを回避でき、パイプラインの効率向上を図ることが可能となる [1][16] [17]。

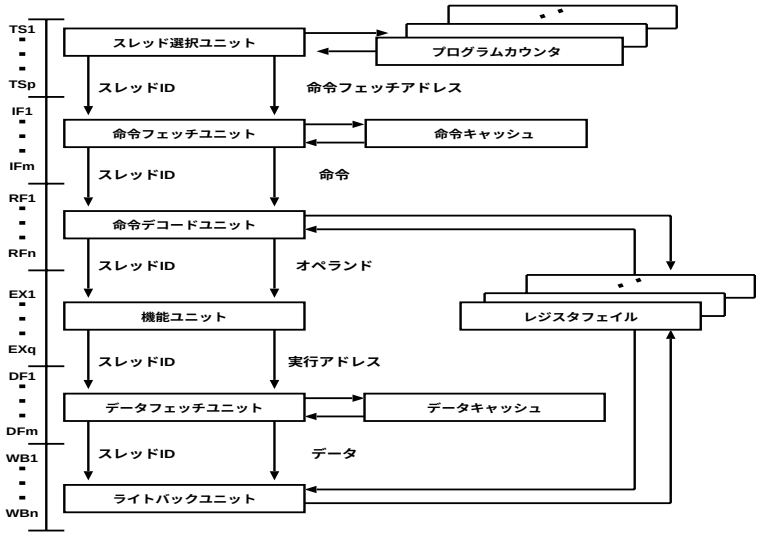

図 5.1: ハードウェア構成

5.3 関数型言語とマルチスレッド処理

関数型言語は式の集まりであり、各々の式は比較的に小さいプロセス（スレッド）に相当し、並列に処理可能な非常に多数のスレッドをパイプラインに投入できる。従ってパイプラインに投入出来るスレッド数が少ないことによるマルチスレッド処理の効率低下を生じることはない。その上に、頻繁な関数呼び出しによる、パイプラインを乱れもこのプロセッサではストールが生じない。

以上のことから、関数型言語とマルチスレッド処理を組み合わせれば、実行サイクル数は大幅に短縮され、同時にストール隠蔽効果により、パイプラインのスループットが増加することが可能になる。このことを証明するために、最初は単純な構成のマルチスレッドパイラインプロセッサを設計する。次に、単純なサンプルプログラムをハンドコンパイラし、その命令列をパイプラインで流すことにより評価を行う。

CPU は、小さな手順の連続という形で個々の命令を実行する。それらの手順は次のようなものである。

1. メモリから命令レジスタに次の命令をフェッチする。
2. 次の命令を指すようにプログラムカウンタを書き換える。
3. フェッチしたばかりの命令のタイプを判定する。
4. 命令がメモリ内のワードを使う場合、その位置を判定する。
5. 必要なら、メモリ内のワードデータを CPU レジスタにフェッチする。
6. 命令を実行する。
7. 手順 1 に戻り、次の命令の実行を開始する。

この手順の連続は、フェッチ-デコード-実行 (fetch-decode-execute) サイクルと呼ばれる。

これは、すべてのコンピュータの中核である [20]。これらの命令の段階 (ステージ) はプロセッサ高速化技術の鍵となるパイプラインで実行できる。従って、最も基本的なパイプラインのステージ数は 3 段からなり、また、マルチスレッド処理によるパイプライン · ストールおよび同期処理などの隠蔽効果を知るには最低 3 段からなるマルチスレッドパイプラインプロセッサが必要である。

もっとも高いスループットを得るために、パイプラインの段数をもっと増やす必要がある。しかし、パイプライン段数を増やすほど性能が向上するわけではない。パイプライン処理によって得られる性能向上は３つの要因によって制約される。最初に、プログラム中のデータ · ハザードのため、パイプライン段数を増やせば増やすほど１命令当りの平均実行が増大する。第 2 に、制御ハザードのため、パイプライン段数を増やせば増やすほど分岐が遅くなり、結果としてプログラム実行に要するクロック · サイクルが増大する。最後に、パイプライン · レジスタのオーバヘッドが加算されるので、パイプライン段数を増やしたからといって、クロック周期をどこまでも短くできるわけではない [19]。ここでは、MIPS のパイプライン [19] のように段数を５段にする。

５段からなる通常のパイプラインプロセッサと、５段からなるマルチスレッド型パイプラインプロセッサでそれぞれの効果を測定する。次の章では作成したプロセッサを用いてデータを収集する。

5.4 通常のパイプライン方式

前の節で設計方針を説明した。この節では設計方針によって通常のパイプラインプロセッサを設計する。仕様は以下の通りに約束する。

1. 命令長はすべて 32 ビットにする。
2. 命令メモリとデータメモリを別にし、命令のフェッチと、メモリからのデータの読み出し/への結果の書き戻しを同時することが可能。
3. レジスタは 32 ビットで、汎用レジスタ数は 8 本にする。それぞれは、
   - R0:作業用にする。
   - R1,R2,R3,R4:関数の操作の時、必ず退避、回復するレジスタである。
   - R5:スタック · ポインタ - SP。
   - R6:フレーム · ポインタ - FP。
   - R7:静的リンク用ポインタ - EP。

このプロセッサが実行できる命令セットを次の表 5.1 に示す。表 5.1 の中で R_n はレジスタを、imm16 は 16 ビットの即値を表す。また、パイプラインでハザードが生じる場合は、何もしない命令 NOP 命令を投入することにする。

ハードウェアの構成は図 5.2 に示す。

命令の種類	例	動作の意味
特殊命令	nop	no operation
データ転送命令	mov Rn,Rm	Rn ← Rm
	movi Rn,imm16	Rn ← imm16
	mov #n(Rn),Rm	M((Rn)+n) ← Rm (*n はオフセット*)
	mov Rn,#n(Rm)	Rn ← M((Rm)+n) (*n はオフセット*)
論理演算命令	or Rn,Rm	Rn ← Rn | Rm
	and Rn,Rm	Rn ← Rn & Rm
算術演算命令	inc Rn	Rn ← Rn + 1
	dec Rn	Rn ← Rn - 1
	add Rn,Rm	Rn ← Rn + Rm
	addi Rn,imm16	Rn ← imm16
	sub Rn,Rm	Rn ← Rn - Rm
	cmp Rn,Rm	Rn - Rm
シフト命令	shr Rn	Rn ← Rn ÷ 2
	shl Rn	Rn ← Rn × 2
制御転送命令	bnz Rn,imm16	if R≠0 then pc ← imm16
	bz Rn,imm16	if R = 0 then pc ← imm16
	jmp imm16	pc ← imm16
	jmp Z,imm16	if Z,then pc ← imm16
	jmp NZ,imm16	if not Z, then pc ← imm16
	jr Rn	pc ← Rn
	jal Rn,ProcedureAddress	Rn ← pc+4; go to ProcedureAddress

表 5.1: パイプライン処理に必要な命令セット

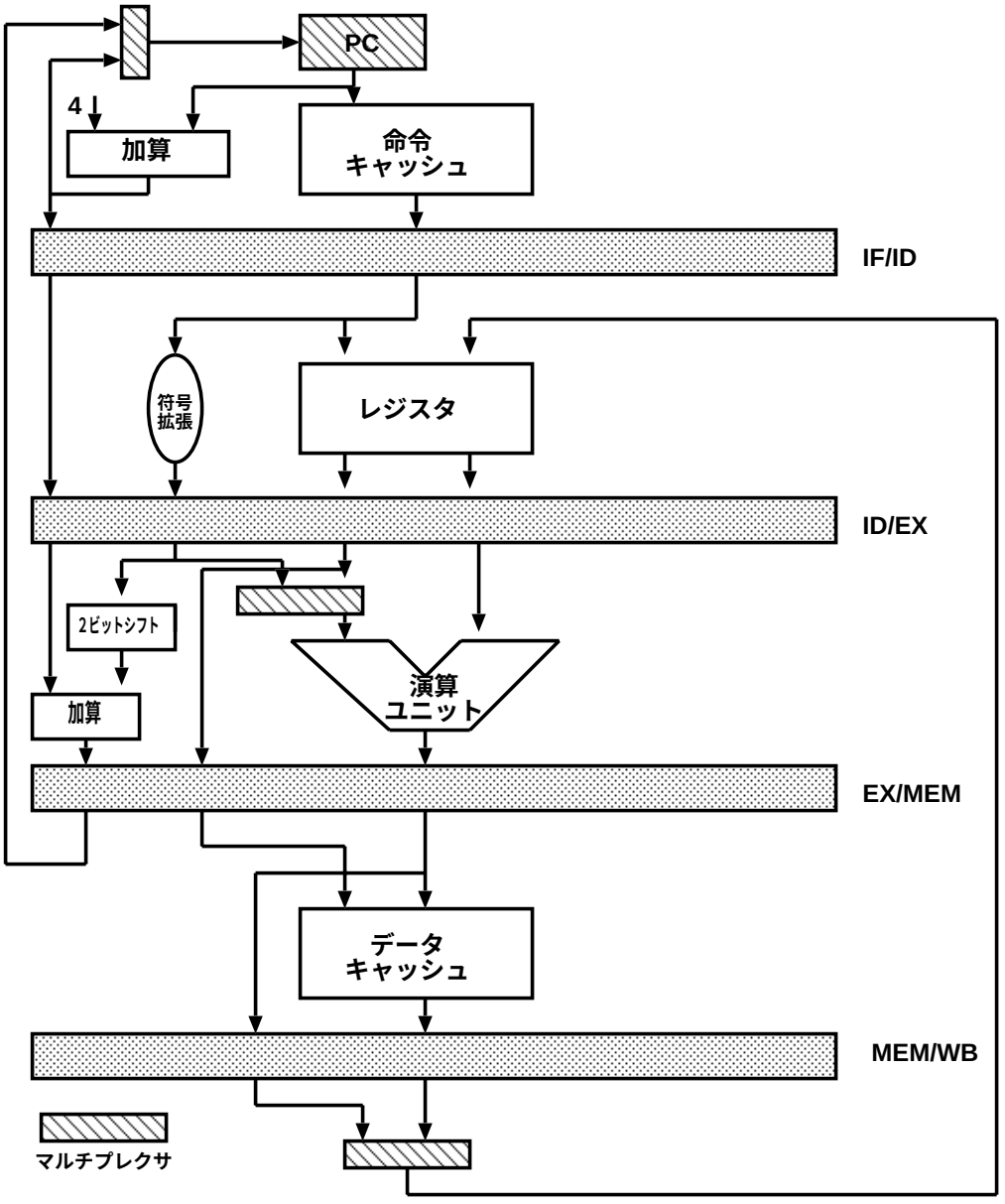

図 5.2: 通常のプロセッサデータパスのパイプラインの構成

5.5　マルチスレッド型方式

前の節と同じように、この節ではパイプラインを有するマルチスレッド·パイプライン·プロセッサを設計する。仕様と約束をは以下の通りにする。

1. 命令長はすべて 32 ビットにする。
2. レジスタは 32 ビットで、汎用レジスタ数は 8 本にする。それぞれは、
   - R0,R1:作業用にする。
   - R2,R3,R4,R5:関数の操作の時、必ず退避、回復するレジスタである。
   - R6:静的リンク用ポインタ - EP。
   - R7:フレーム·ポインタ - FP。
3. レジスタ数＝ステージ数 × 汎用レジスタ 8 本。レジスタ使用の規約は通常のパイプラインの同一とする。
4. 命令メモリとデータメモリを分割し、命令フェッチと結果のメモリへの書き込みは同時に可能。
5. マルチスレッド型プロセッサは毎サイクルに異なるスレッドを実行するので、メモリアクセスの局所性がなくなる可能性があり、キャッシュメモリのミス率は上昇する可能性がある。ここでは、メモリとキャッシュがかなり巧みに作られ、ミス率が上昇しないと仮定する。

このプロセッサで実行できる命令は表 5.1、表 5.2 に示す。表 5.2 は、投入する複数のスレッドを制御するために追加した。パイプラインを 5 段にして、５つの命令をオーバーラップして発行することができる。この異なる５つのスレッドをパイプラインするために、スレッド数と同数のレジスタ、プログラムカウント、フラグを用意する。追加した命令の中、movTh 命令は異なるレジスタ間のデータ転送命令であり、スレッド R_k の R_i に現在のスレッドのレジスタ R_j のデータを入れる。setPC 命令は、スレッド R_n の PC を imm16 に設定する命令である。opnT 命令は、レジスタ Rn の新しいスレッドをスタートさせるために用意した命令である。この命令が実行させると、スレッドが使用中であることを示すフラグをセットし、パイプライン処理で使用中のスレッドがどっちか知ることができる。この命令と逆に、clsT 命令を用意する。これはスレッドのフラグをクリアさせ、命令フェッチの停止やスレッドの使用終了通知を可能にする。もし、Rn=0 の場合は、スレッドの自己終了する。

マルチスレッド型プロセッサのパイプラインの構成は図 5.3 に示す。図中の点線で囲まれた部分がマルチスレッド処理をするために追加した機構である。

マルチスレッド型プロセッサでは、複数個のスレッドが存在し、各スレッドは危険区域 (critical section) と呼ばれるコード部分を持ち、共用のリソースの変数を読んだり、更新したりする。この場合では、スレッドの間で排他的制御 (exclusive control) を実現しなければならない。つまり、メモリへのアクセスを唯一のスレッドからのみ許せなければならない。

命令の種類	命令の例	動作の意味
スレッド間のレジスタ転送命令	movT R_k, R_i, R_j	Thread R_k の $R_i \leftarrow R_j$
PC セット命令	setPC R_n,pc(FP)	Thread R_n の pc $\leftarrow$ pc + FP
スレッドの get	getT R_n	$R_n \leftarrow$ ThreadNumber
スレッド開始命令	opnT, Rn	Open Thread (Rn),T(Rn)=1
スレッド終了命令	clsT, Rn	Close Thread (Rn),T(Rn)=0 Rn=0 の場合はスレッド自己終了である。

表 5.2: マルチスレッド処理に必要な命令セット

マルチプロセッサの場合は、プロセッサが独立していますから、この排他制御は、メモリのロック線により行うことができる。つまり、メモリのアクセスの時にはロック線をアサートし、他のプロセッサからアクセスを拒否する。メモリのアクセス動作が終了するとロック線を解放し、他のプロセッサからのアクセスを認める。

排他制御を実現するために、マルチスレッドプロセッサの場合はメモリの参照命令にロック/アンロックの機能を追加する必要がある。次のようにメモリ参照命令にロック機能を追加する。

movl R_j , #n(R_i)
　　排他制御されたリージョン
movul #n(R_i) , R_j

しかしながら、このメモリロックのフラッグレジスタを設ける方法では、メモリ全体をロックしてしまうので、効率が悪くなる。このために、ロック対象はメモリアドレスに対してだけにする。ロック対象アドレスを仮想記憶に記憶して置き、ロック付きメモリアクセス命令 (movl,movul) では、この仮想記憶を参照して、排他制御操作を行う。

- movl 命令
  1. アドレス #n(R_i) が仮想記憶にない時、ロック成功し、仮想記憶にアドレスを登録する。ロックフラッグを立てる。
  2. アドレス #n(R_i) が仮想記憶にある時、ロック不成功し、スレッドの実行を中止する。パイプラインを空送りする。

- movul 命令
  1. アドレス #n(R_i) が仮想記憶にある時、ロックを解除し、仮想記憶にアドレスを消し、ロックフラッグを消す。
  2. アドレス #n(R_i) が仮想記憶にない時、NOP

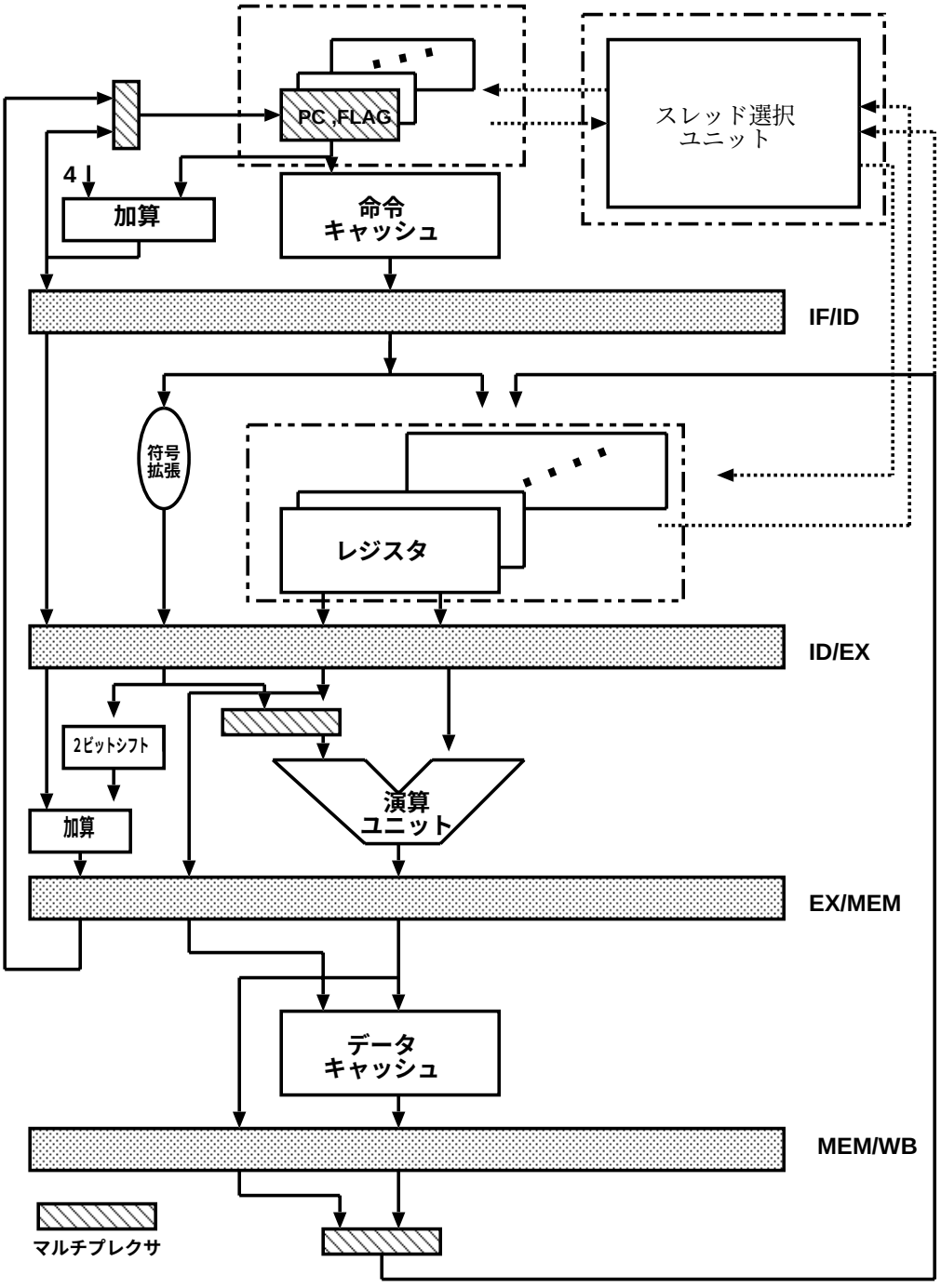

図 5.3: マルチスレッドデータパスのパイプラインの構成

第6章　提案する手法及び評価

6.1　データ依存関係の検査方法

SML の式は、いつもある環境の元で式の評価を行う。環境とは、値によって束縛された変数の集合である。したがって、式は環境によって評価される。SML では、静的スコープ規則を持つ言語であり、変数の値は、その変数が定義された時点の値に固定され、以後は変化することはない。この規則は関数定義にも適用される。関数本体の束縛変数以外の自由変数、それらはの変数の意味は、関数が定義された時点でのその変数に束縛されている値であり、関数が実行される時点での同一名の変数の定義とは関係ない。

従って、本研究では環境に注目しながら、データ依存関係の検査をする。コンパイルの実装の時、フロントエンドでの記号管理表の実装で、値の項目にフラッグをつけることに解決を図った。変数が定義済みの場合、Flag に 1 を設定、未定義の場合は Flag に 0 を設定する。処理段階では記号管理表を調べて、その Flag を検査して、変数の定義済みか否かを判断することにより、データ依存関係をチェックする。

6.2　式の間での同期処理

式の実行時間が種々であり、式の間で、並列処理をした後、同期を行う必要がある。式の値が、部分式の評価が完全に終わる前に値を返すと正しくない結果を招く。そのような誤りを避けるために、本研究では同期処理を以下のように行う。

式の全体を評価する時、式の要素 (**部分式**) の評価はその式の値と完了状況を知るマークを一緒に返すことにする。要素が全完了したら、1 を、そうではない時は 0 を返すようにする。例えば、次のような式を評価する時、

val Exp = exp_1 + exp_2 + exp_3;

exp_1;exp_2;exp_3 が並列実行可能なら、同期処理はマーク A,B,C をチェックすることによって判断する。図 6.1 は、提案する方法による上の式の同期処理の説明図である。

start exp_1; exp_2; exp_3;
val A = 0 ; B = 0 ; C = 0;
val Exp[A,B,C] = case [A,B,C] **of** [1,1,1]
=> Evaluation
| _ => Suspension

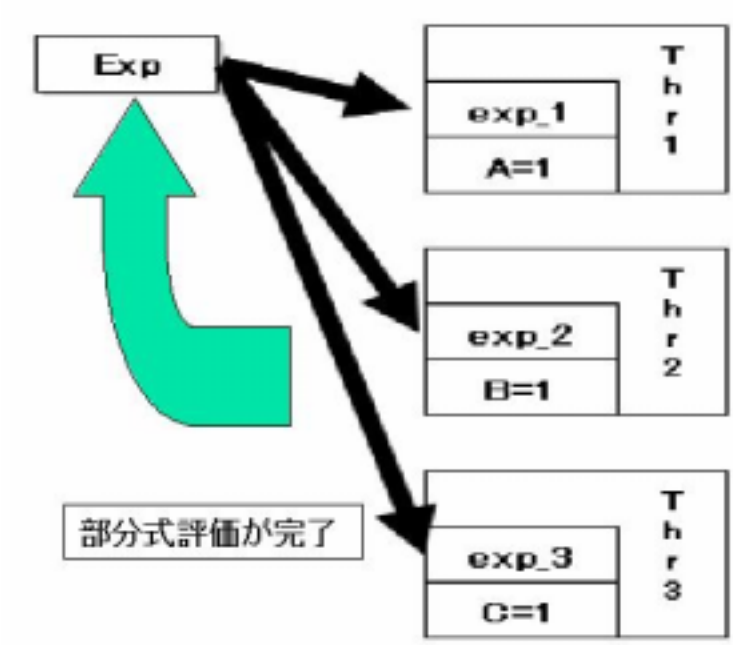

図 6.1: 関数合流のための同期処理

6.3 評価

3.1.3 節で述べたように、関数型言語は頻繁な関数の呼び出しによる、スタック上のフレーム操作の命令により、効率がよくない。スタックフレーム操作処理は、(1) 関数の呼び出しの時、(2) 関数からの復帰時に必要となる。

6.3.1 フレーム操作

1. 関数の呼び出しの時、
   以下のような操作が必要である。
   (a) 動的リンク (FP)、静的リンク (EP) の退避場所を確保、そのために、引数をスタック上で上部にスライドさせる。
   (b) FP、EP の退避と、FP の更新を行う。
   (c) レジスタの退避を行う。
   (d) PC の退避、つまり戻りのアドレスを記憶する。
   その操作を次のような命令列で表現できる。スタックの進行状況は図 6.2 のようになる。

```
%1 mov R0,0(SP) ; # slide arg-2
%2 mov 8(SP),R0 ;
%3 mov R0,-4(SP) ; # slide arg-1
%4 mov 4(SP),R0 ;
```

```
%5 mov -4(SP),EP ; # 環境ポインタをプッシュ(静的リンク)
%6 mov 0(SP),FP ; # フレームポインタをプッシュ(動的リンク)
%7 mov FP,SP ; #  set new FP

%8 addi SP,8 ; # SP:=SP + 8
%9 mov 0(SP),R1 ; # PUSH R1
. . .
. . .
%10 mov 4i(SP),Ri ; # PUSH Ri
. . .
. . .
%11 mov 4n(SP),Rn ; # PUSH Rn
%12 addi SP,4n ; # SP:=SP + 4n
%13 jal fun ; # 戻りアドレスを PUSH、関数を呼び出し。
```

この命令列の中での命令セットは表 5.1 の通りである。

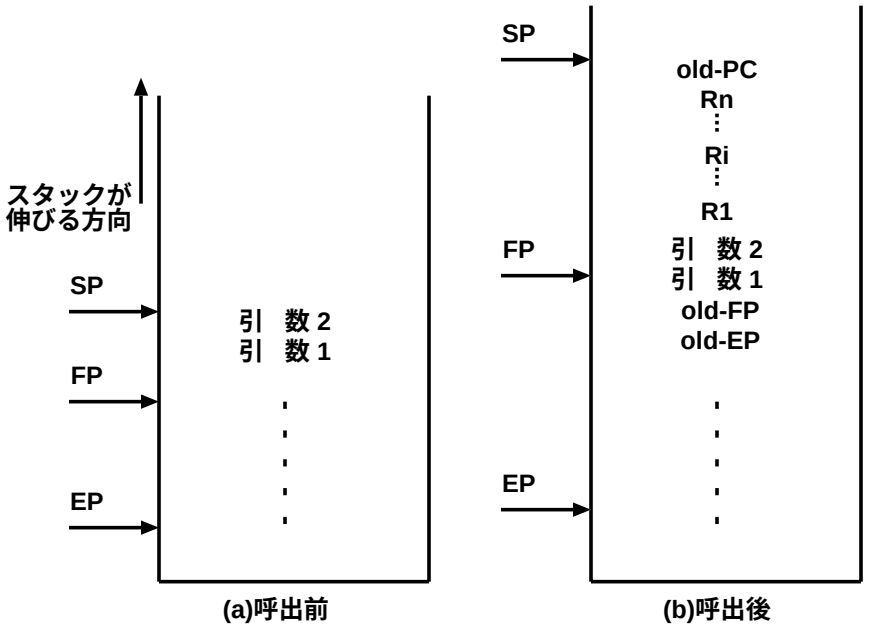

図 6.2: スタックフレームの操作進行状況

2. 関数からの復帰の時
 (a) EP を元の値に戻す。
 (b) 関数の結果をスタックの所定の所にスライドする。
 (c) レジスタを回復する。

(d) FP を元の値に戻す。
(e) リターンする。

```
%1 mov EP,-8(FP) ; # EP を元の値に戻す。

%2 mov R0,0(SP) ; # 関数の結果。
%3 mov -8(FP),R0 ; # 関数の結果をスライドする。

%4 mov R1,8(FP) ; # レジスタを回復する。
%5 mov R2,12(FP) ; #
. . .
. . .
%6 mov Rn,4n+8(FP); #
%7 mov R0,4n+12(FP); # 戻りのアドレスを回復する。

%8 mov SP,-8(FP) ; # SP を回復し、関数の結果を指すようにする。
%9 mov FP,-4(FP) ; # FP を回復
%10jr R0 ; # 関数の呼び出し側に戻す。
```

3. 通常パイプラインの評価
   以上の２つの議論から分かるように、通常のパイプライン処理では関数の呼び出しよる、レジスタとメモリの間に退避と回復が非常に頻繁に起きる。(1) で議論した命令列の%5 から%13 間の部分である。それらが、パイプラインストールの原因になる。特に、関数型言語は関数の呼び出しが非常に多いので、通常のパイプラインでは効率がよく落ちる。

6.3.2 マルチスレッドプロセッサ

マルチスレッド型プロセッサはスレッド毎に、自分専用のレジスタを持ち、関数の呼び出しが起っても、レジスタの退避は行わないので、ハザードが起きない。ただ、関数がデータ依存関係のために、suspend される場合はレジスタの内容をメモリに退避する必要がある。または、スレッドの再開の場合でも、メモリの内容をレジスタに回復を行う必要がある。その時、他のスレッドは影響を受けず走ることができ、その分によりオーバラップさせる効果によりパイプラインのステージを埋めることが可能となる。つまり、スループットという概念で言えば、通常のパイプラインよりかなり高効率が得られる。

マルチスレッド型プロセッサでは、関数の呼び出しによりスレッドを生成、処理し、結果の戻しは次のような手順で行う。

1. まず、関数の呼び出しより、スレッドを生成し、TCB(Thread Control Block) を生成する。TCB はスレッドフレーム (Thread Frame) と呼ぶ。

2. 生成された TCB に引数を書き込む。
3. 戻り番地と親スレッドのアドレスの番地を設定する。
4. 子のスレッドを生成する時、親のスレッドは子に対して、親の場所と子スレッドの結果を格納するための親スレッドと、親スレッドでの位置 (オフセット) の組を子に渡す必要がある。子スレッドが結果を得るとその (親のポイント、オフセット) により、結果の書き込み場所を計算する。
5. suspend し (子スレッドからの結果が揃うのを待つため)、suspend 処理を行う。
6. 結果が子のスレッドから親に戻って来ると、親のスレッドはアクティブになり、親スレッドは直ちに実行する (resume) か、もしパイプラインがいっぱいの場合はスレッドを ready queue に入れ、後で実行する。

suspend 処理は次のように行う。

1. レジスタの内容を TCB に退避する。PC も退避する。
2. 子のスレッドの生成処理に戻る。

resume の処理は、TCB の内容をレジスタにセットし、PC もセットし、スレッドを open して実行を開始する。

上の通常のパイプラインと対応する、関数の呼び出しとフ復帰の処理次のように展開される 。

[関数の呼び出しシーケンス]

```
% movl R1, Thpool;  Thpool に新しい FP(new_FP) が入っているとする。
%
% mov R0, arg_1(FP); 引数 1 をスレッドにコピーする。
% mov arg_1(FP),R0;
% mov R0, arg_2(FP); 引数 2 をスレッドにコピーする。
% mov arg_2(FP),R0;
%
%戻り番地と親スレッドの番地を設定
% mov R0,#return_value;
% mov return_to(R1),R0;
% mov return_to+4(R1),FP;
%
% suspend-count を設定する
% mov R0,#1;
% mov scount(FP),R0;
%
% レジスタの退避を行う。
```

```
% mov regs(FP),R2;
% mov regs+4(FP),R3;
% mov regs+8(FP),R4;
% mov regs+12(FP),R5;
%
% PC を設定する。
% jal R0,next; R0 に関数の戻りアドレスが入る。
next:
% mov pc(FP),R0; 戻りアドレスを自分のスレッドの中に設定
% clsT ;
%関数の呼び出しの終了し、スレッドを close する。
```

[**関数からの復帰シーケンス**]

```
% Ri <-- 結果が入っているとすると。
% mov R1,return_to(FP);
% mov R2,return_to+4(FP);
% add R1, R2;
% mov (R1),Ri; 値を書き込む。
% movl R0,scount(R2);
% dec R0;
% bnz notresume;
%
% resume の開始。
% getTh R0; スレッドの get
% bz nothread ready_queue ; thread がないと queue に飛ぶ。
% mov R1,regs(FP); レジスタ R2 を回復する。
% movT R0,R2,R1; スレッド R0 のレジスタ R2 に R1 を書き込む。
% mov R1,regs+4(FP);
% movT R0,R3,R1;
% mov R1,regs+8(FP);
% movT R0,R4,R1;
% mov R1,regs+12(FP);
% movT R0,R5,R1; レジスタの回復の終了。
%
% setPC R0,PC(FP); プログラムカウンタをセットする。
% opnT R0; スレッドを開始する。
% clsT ; スレッドの終了。
```

```
%
notresume:
% clsT ;
%
ready_queue:
% movl R1,Qtail;
% movl (R1),FP;
% movul Qtail, FP;
% clsT ;
%
% 関数の復帰の終了。
```

そのTCB、スレッドスフレームは図6.3のように構成されている。

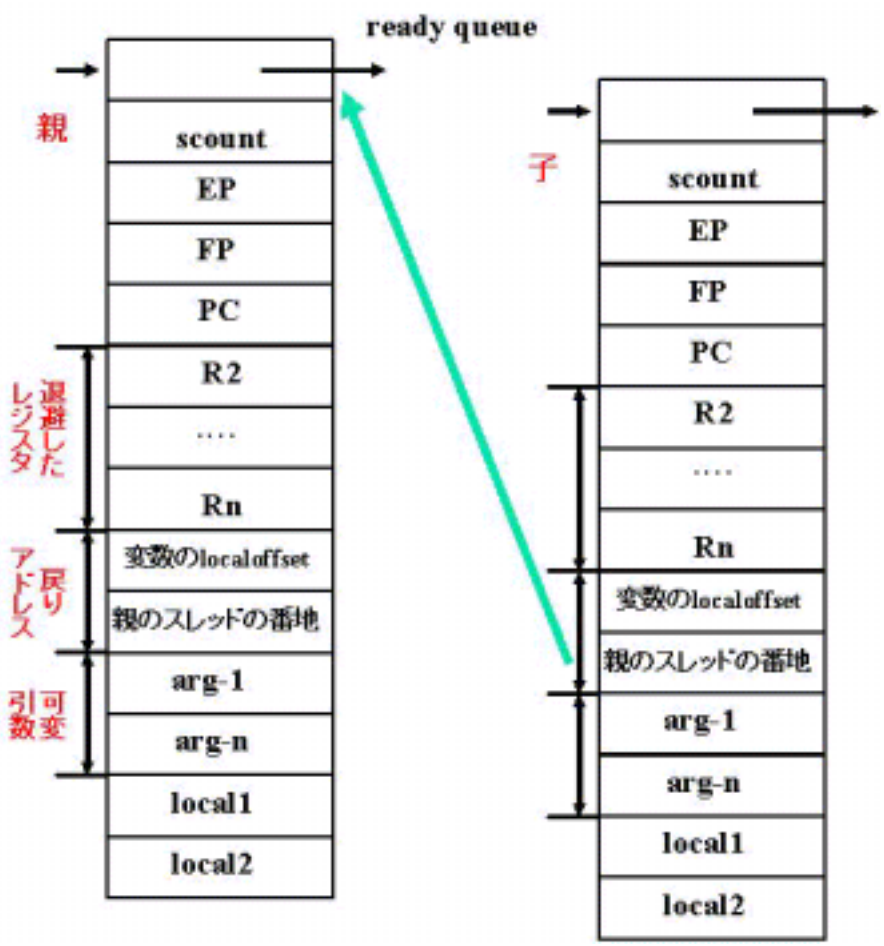

図 6.3: スレッドフレームの構成

上に示した例により、関数の呼び出しと復帰の処理を通常パイプラインとマルチスレッ

ド型プロセッサとで比較することにより、理論的にマルチスレッド型プロセッサの実行の高効率が証明できる。マルチスレッド型プロセッサのほうは、命令ステップ数が多いが、ストールがなくなる。また、ある関数の suspend が起きても、その分のオーバヘッドはほかのスレッドの実行により隠蔽できる。

命令列のステップ数で簡単な評価が得られる。以下の仮定をして単純な評価をする。

1. マルチスレッド型プロセッサは通常のパイプラインより CPI が 30% 小さい。
2. プログラムの中で関数の呼び出しと復帰処理のステップの割合がが 50 %。

その仮定の上で、前の２つの命令列の例からそれぞれの実行のステップ数を計算する。通常パイプラインの場合は、関数呼び出しと復帰処理の命令の総ステップ数は 25 ステップである。マルチスレッドの場合、３つのケースがあり、それぞれは 26、31、38 ステップである。マルチスレッドが最悪の場合、ステップ数が一番長い 38 ステップの時 CPI の比で性能を比較すると、$CPI_{multi}/CPI_{normal} = 0.7 * 0.5 + 0.7 * 0.5 * (38 / 25) = 0.882$ であり、通常のパイプラインより、11.8%早くなることが期待される。

つぎのようにケースを増やして、評価をすると図 6.4 のようになる

- マルチスレッド型プロセッサが通常のパイプラインより CPI が 10%,20%,30%,40%,50%,60% 小さい。
- プログラムの中で、関数の呼び出しと復帰処理が 50% 占める。

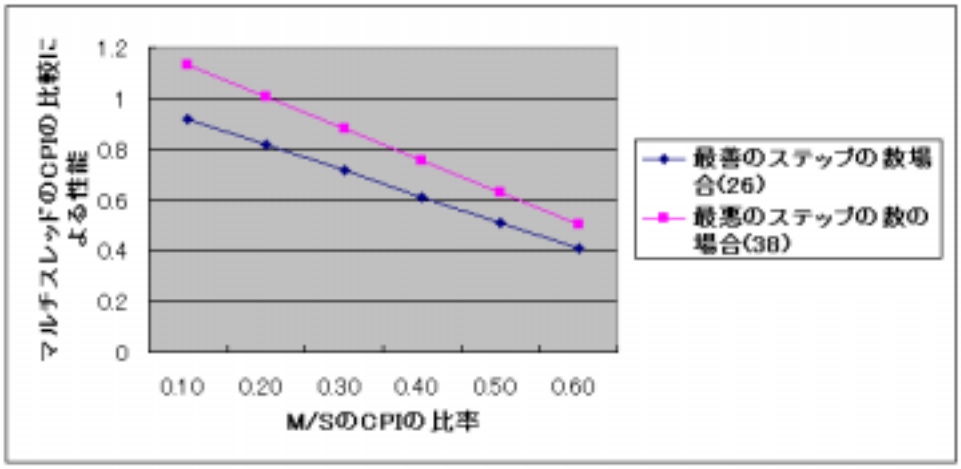

図 6.4: マルチスレッド型プロセッサの CPI による性能

第7章　結論

まず第2章では、マルチスレッド型プロセッサを論じ、そのアーキテクチャがどのような特徴を持っているかを述べる。

次に第3章では、関数型言語がどのようなものであり、またどのように処理されるかを述べ、パイプライン実行上での問題点を論じる。

第4章では、評価を行う前段階として、コンパイラの実装方法を述べ、この方法従って、実装を行う。第5章では、提案した手法の効果を評価するために、通常パイプラインとマルチスレッド型プロセッサのアーキテクチャを構成する。

また章では、実際の評価し、得たデータにより、提案手法の有効性を議論する。

最後の章では、本研究の結論を述べるとともに、今後の課題を論じる。

本論文では理論的にマルチスレッド型プロセッサと関数型言語の特徴を組み合わせると高効率が得られることを示したのみで、、シミュレーションによる定量的な評価は行っていない。今後の課題として、理論的に証明したことを確認する必要があり、主に以下のことが考えられる。

1. スレッドの導入により、再開とクローズ専用命令を使った。この命令の追加による、オーバヘッドを考えるべきである。
2. 停止されたスレッドを再開する時、どのスレッドを選択するか、すなわちスレッドスケジューリングの損失も考えるべきである。
3. マルチスレッド型プロセッサは毎回異なるスレッドをパイプラインに投入し、それによりメモリ、キャッシュの空間的な局所性がなくなり、メモリのアクセスミス率が上昇すると考える。シミュレーションする時、このキャッシュミス、メモリのペナルティの影響も考えるべきである。
4. 今回コンパイラの実装を行なったが、コード生成の所まで及ばなかった。今後は、命令生成部の実装を行い、様々なテストプログラムにおいてマルチスレッド型プロセッサの有効性を実証していく必要がある。

謝辞

本研究を行うにあたり、終始熱心かつ寛容な御指導・御鞭撻を賜りました日比野 靖 教授に深く感謝致します。また、本研究を進めることにあたり、有益な御助言を頂きました堀口 進教授、田中 清史 助教授に心より感謝致します。排他的制御の問題点をご指摘を戴きました篠田陽一教授にも深く感謝致します。

マルチスレッド型アーキテクチャの構成の際に伊藤さんの研究からのヒントと、命令セットの規定と命令コードの生成の際に役に立った細井さんのヒントに対しても心より感謝致します。

最後に、本研究の過程において、いつも励まして下さった日比野研と田中研のみなさんに心から御礼を申し上げます。

参考文献

[1] 日比野 靖、“マルチスレッド型超パイプラインプロセッサアーキテクチャ”、科学研究費補助金基盤研究（**B**）(2) 研究成果報告書、2001。

[2] 日比野 靖、“高級言語計算機”、電子情報通信学会誌、Vol.78,No.11,pp.1164-1170,1995年。

[3] 大堀 淳、“プログラミング言語 ML-その歴史、特徴、応用および最近の研究動向-”、情報処理学会誌 「情報処理」 35 巻、3 号、1994 年。

[4] 雨宮真人、田中譲、“コンピュータアーキテクチャ”、オーム社、1988。

[5] Michael R.Hansen and Hans Risechel, “Introduction to programming using SML”、Addison Wesley、1999。

[6] 大堀 淳、“Standard ML 入門”、共立出版株式会社、2001 年。

[7] R. バード、P. ワドラー共著、武市正人訳、“関数プログラミング”、近代科学社、1991年。

[8] A.V. エイホ、R. セシィ共著、“コンパイラ I、II”、株式会社サイエンス社、1990 年。

[9] 原田賢一、“コンパイラ構成法”、共立出版株式会社、1999 年。

[10] 中田育男、“コンパイラの構成と最適化”、朝倉書店、1999 年。

[11] 疋田輝雄、石畑清共著、“コンパイラの理論と実現”、朝倉書店、1999 年。

[12] Andrew W.Appel, “modern compiler implementation in ML”、CAMBRIDGE UNIVERSITY PRESS、1998 年。

[13] John H. Reppy, “Concurrent Programming in ML”、CAMBRIDGE UNIVERSITY PRESS、1999 年。

[14] http://www.smlnj.org//index.html

[15] http://www.standardml.org/Basis/

[16] 細井雅之、“並列論理型言語の実行に適したマルチスレッド型プロセッサアーキテクチャに関する研究”、北陸先端科学技術大学院大学修士論文、2002 年。

[17] 伊藤英治、“関数型プログラムの実行に適したマルチスレッド型プロセッサアーキテクチャに関する研究”、北陸先端科学技術大学院大学修士論文、1997 年。

[18] David A. Patterson and John L.Hennessy, “Computer architcture:A quantitative approach”,Morgan Kaufmann Publishers,Inc.,1990. 邦訳：富田真治、村上和彰、新実治男、“コンピュータアーキテクチャ：設計、実現、評価の定量的アプローチ”、日経 BP 社、1999 年。

[19] John L.Hennessy and David A.Patterson, “Computer Organization & Design:The Hardware/Software Interface Second Edition”, Morgan Kaufmann Publishers,Inc.,1998. 邦訳：成田光彰、“コンピュータの構成と設計 第 2 版 「上」「下」：ハードウェアとソフトウェアのインターフェース”、 日経 BP 社、1999 年。

[20] Andrew S.Tanennbaum, STRUCTURED COMPUTER ORGANIZATION,Prentice Hall,Inc.,1999. 邦訳：長尾高弘、構造化コンピュータ構成、株式会社ロングテール、2000。

[21] ディビッド・クラーク、ドナルド・メルシ共著、小松 伸行訳、“システムソフトウェアプログラミング–コンパイラの設計法と並行処理の基礎”、株式会社パアソン・エデュケーション、1998 年。

[22] Bil Lewis、Daniel J.Berg 共著、岩本信一訳、“マルチスレッドプログラミング入門”、アスキー出版局、1996 年。

[23] 久野 靖 著、“言語プロセッサ”、丸善株式会社、1993 年。

[24] J. ピーターソン、A シルバーシャッツ共著、宇津宮孝一、福田晃共訳、“オペレーティングシステムの概念”、原書第 2 版、培風館、1987 年。

[25] 中澤喜三郎著、“計算機アーキテクチャと構成方式”、朝倉書店、1995 年。