言語処理系における意味処理の並列化に関する研究

1992年

西山博泰

①

言語処理系における意味処理の並列化に関する研究

1992年

西山 博泰

目次

第 1 章

はじめに

コンピュータのハードウェアの分野では，プロセッサの高速化，メモリの大容量化が進んでおり，これに伴ってソフトウェアも大規模なものが許容されるようになってきた．このため，プログラムを開発するコストの中で，コンパイルによって占められる割合も無視できなくなりつつある．また，近年のユーザ・インターフェイスの発展と普及により，近年のソフトウェア開発は大規模化は拍車を掛けられる方向にある．例えば，X ウィンドウ・システム [Scheifler 86] や emacs[Stallman 86] などでは，システムの再構築のためのコンパイルだけでも，数時間のオーダの時間が必要である． make[Feldman 86] 等のプログラム構成支援ツールの利用やプログラムの一部のライブラリ化により，こういった問題の一部については解決する事が可能であるが，これは本質的な問題の解決にはつながらない．このような状況では，編集，コンパイル，実行とデバッグが繰り返されるプログラムの開発サイクルにおいて，プログラムの編集やデバッグに要する時間に比べ，プログラムをコンパイルするための時間が相対的に増加し，プログラム開発者がプログラム自身に専心することを妨げる結果となり，言語処理系の高速化が重要な課題となる．

言語処理系高速化のための技法としては，インクリメンタル・コンパイルに基づく方式 [Earley 72] が研究されてきた．この方式の実現には，コンパイラとエディタが密に結合され，プログラムの中間的な表現とオブジェクト・コードとの対応表を持つ必要があるなどの点から広く普及するには至っていない．また，中間的な表現がツール間で共有されることからユーザに対してエディタなどのユーザインタフェースが固定されるなどの問題がある．このような点から，言語処理系自身の高速化は，現状のプログラミング環境の延長上に存在し，ユーザに対して劇的なプログラミング環境の変化や，複数の環境の使い分けを強いないという点で 1 つの望ましい方向である．

また，言語処理系における様々な技法 [Aho 86][Fischer 88][Sassa 89] は，文脈自由文法で定義可能な構造を持ったデータに対する仕様の記述や，構造を持ったデータに対する操作の適用の具体例と捉えることも可能である．このような観点から，在庫管理システムやファイルシステムの記述への言語処理系の技法の適用 [Katayama 85][Shinoda 86] や，階層的な VLSI デザインシステムへの言語処理系の技法の応用 [Jones 86] がなされている．また，近年では言語指向エディタ [Reps 84]，ユーザインターフェイ

スの記述 [Barford 89]，アルゴリズムアニメーション [Brown 87] など，従来のバッチ的な利用の範疇に収まらない多様で複合的な言語処理系技術の利用形態が現れている．こういった対象に対してはコンパイラなどを対象として研究が行なわれてきた言語処理系技術では対応が難しい場合も多い．このような観点から言語処理系の並列化方式を考えることは，構造を持ったデータに対する並列な処理の適用や分散データの処理の統一的な記述の枠組を考えることへも繋がる．

以上のように，言語処理系における並列処理方式の研究は言語処理系の高速化という観点とともに言語処理系記述の記述性や応用の拡大という 2 点で重要であると考える．このため，本研究では並列実行を基本とした広範囲の応用に適用可能な言語処理系記述と実行の枠組を研究することを目的とし，具体的に言語処理系の意味処理に焦点を当て，その並列化のための方式とマルチプロセッサ上での実行方式の研究を行なった．

1.1 言語処理系における並列処理の諸方式

言語処理系における並列化の導入という観点からの研究は，並列な構文解析法 [Fischer 75][Cohen 85] などに端を発し，コンパイラ全体への並列処理の導入 [Bare 77] や，意味処理の並列実行 [Seshadri 88][Kuiper 90][Jourdan 91]，コード生成の並列化 [Krohn 75] など様々な方向からの研究が行なわれてきた．Gross らによる分類法 [Gross 89] に基づいて，言語処理系における基本的な並列処理方式を分類すると，データ分割 (data partitioning)，パイプライン (pipelinning)，計算分割 (computation partitioning) の 3 つのタイプに分類することができる．

(1) データ分割

データ分割による方式は，入力を分割し，分割されたデータにたいして逐次的で均一な処理を適用する方式であり，SIMD 型の並列システムに対応していると捉えることもできる．この方式は，比較的独立性の高いデータに対して有効なものである．プログラムを構成するモジュールのコンパイル処理を独立に行なう parallel make[Baalbergen 88][Stallman 91] は，この典型的な例であるが，この方式では得られる並列性が，プログラムのモジュール構成やその依存関係によって制限を受けることになる．また，分割コンパイルを支援しない PASCAL のような言語に対する適用は有効でない．

parallel make による並列処理方式が手動のプログラム分割による並列化であるのに対して，プログラムを自動的に分割することよる並列化も存在する．一般にプログラムはその構造が意味構造に対応しているため，意味的な構造に対応した部分でプログラムを分割するには何らかの手段で言語構造の認識を必要とする．すなわち，関数や手続きなど構文的に意味を持った対象に対して正確な分割を行なうためには，字句解析および構文解析処理を行なう必要がある．このため，Gross らによる実現では構文の認識と意味チェック処理までを逐次的に行なった後で，関数単位に逐次的な処理の適用を行なっている [Gross 89]．また，オブジェクト・コードの生成処理についても逐次的な処理を行なっている．

これに対して小池らによる実現では，より機械的な入力テキストの分割処理を採用している [Koike 89]．ここでは，コメントや文字列の認識や分割のための境界点認識のため，言語に依存した確率的な予測を行なっている．言語の構造を無視したこのような処理は，対象とする言語の構造を注意深く観察することによって可能となるかもしれないが，これは言語処理系の記述者に大きな負担を課すものであり，言語処理系の構成のための枠組としては問題がある．

最近では高水準のプログラミング言語や，ユーザインターフェイスの定義から対応するC言語などのプログラムを機械的に生成するシステムが数多く構成されている．このようなシステムで生成されたプログラムは大きなものになりがちであり，このような対象に対する有効性には疑問がある．また，データ分割による手法は基本的に，対象言語を特定した手書きのコンパイラを対象としており，広範囲な対象に適用するには不十分である．

(2) パイプライン

パイプラインによる方式は，データの消費と生成を行なう複数の計算を直列的に結合し，独立に処理を行なうものである．まず，並列処理による言語処理系の高速化という観点から，その典型的な例としてコンパイラを考えてみると，コンパイラは通常，字句解析，構文解析，意味解析，コード生成の4つのフェーズ，もしくは，最適化フェーズを加えた5つのフェーズ，と記号表から成るものが一般的である．これらのフェーズは比較的独立性が高く，容易にパイプライン処理による並列処理を行なうことが可能である．また，コンパイラの処理を，字句解析，構文解析，意味解析，コード生成といった論理的な独立性に沿ってモジュールに分割し，各々をパイプライン結合した構成にすることは自然であり，各モジュールの設計や保守が容易になる．モジュールへの分割に際しては，記号表のように一般のコンパイラでは複数のフェーズ間で共有されることの多いデータの扱いが問題となるが，これに関してはこれを各フェーズで必要な情報へと分割し，必要なデータを局所的な情報として各フェーズに持たせることにより、各フェーズを構成するモジュールが独立に動作するようにし，パイプラインによる動作を可能とすることができる [Nishiyama 88a-b]．

ただし，パイプライン構成による並列処理をコンパイラに取り入れただけでは，基本的にモジュール数以上のスピードアップは期待できず，最も遅いフェーズがボトルネックとなってしまう [Koike 89]．通常のコンパイラでは，コンパイラを構成するフェーズのうちで最も負荷の高いフェーズ，すなわち，ボトルネックとなるフェーズは，複雑な最適化を行なわない比較的単純なコンパイラでは字句解析部であり [Wait 86]，最適化等により意味処理の内容が複雑化するにつれ意味解析部の処理がボトルネックとなると予想される．

我々が実際に手元にソースコードがあるコンパイラのうち，教育向けの簡単なシステムであるPASCAL-P4コンパイラ [Pemberton 82] と，実用レベルのコンパイラであるGCC[Stallman 92] に関して最適化を行なった場合と行なわなかった場合の処理における各フェーズの処理時間を計測した結果から，PASCAL-P4のような単純な構

成のコンパイラでは，字句解析処理が実行時間の半分程度を占めており，コード生成処理が1/4程度となっていることが判明している．ここでの各フェーズの処理を詳細に検討してみると，コンパイラを構成するフェーズのうち字句解析，および構文解析処理は単純な処理ではあるが頻繁に呼び出されることから，単純なコンパイラではこれらのフェーズの実行時間が処理の大半を占めていことがわかる．また，GCCでは最適化を行なわない場合でも意味解析部の処理が半分程度，最適化処理を行なった場合，意味解析処理が全体の75%程度を占める結果となっている．我々は，字句解析部や構文解析部を高速化する手法の1つとして，字句解析処理や構文解析を専用に行なうハードウェアを提案しており [Itano 88,89]，これらのハードウェアを用いることにより，幾つかのフェーズに関しては高速化を行なうことが可能である．実際，これらの専用ハードウェアを採用することで，字句解析部の処理については1トークンあたり数クロック，構文解析部については数十クロックで解析を行なうことが可能であることが確認されている [Itano 88][Ng 88]．これはソフトウェアで処理した場合の数十から数百倍の性能であり，PASCAL-P4のような単純なコンパイラの意味解析部と比較しても無視できる程度のスピードである．また，構文解析に関しては，解析表を直接実行する機械命令に変換する方式 [Pennello 86] も知られており，この方式では6から10倍程度の高速化が可能であると報告されている．さらに，コード生成フェーズについても構文解析と同様の手法を用いたパターンマッチによる手法 [Graham 80] や属性文法を用いる方式 [Ganapathi 82] が知られており，その処理の高速化に関しては構文解析処理と意味解析処理の高速化の問題に還元することができる．すなわち，コード生成に関しても基本的には構文解析コプロセッサと同様の手法を用いることで処理を高速化することができる．

以上のように，言語処理系の論理的なフェーズを独立に動作することによってパイプライン型の並列処理を行なうためには意味処理の高速な実現が不可欠である．また，パイプライン型の処理は，基本的には (1) や (3) の方式と直交した概念であり，これらの組合せを行なうことによりモジュール化された言語処理系構成と，並列処理を両立させることができる．

(3) 計算分割

計算分割による方式は，対象に対して必要とされる計算を何らかの手段により分割し，非均等な処理を独立に行なうものである．言語処理系における意味処理の記述とその実行は，一般的なプログラミング言語としての側面と共に，対象の構造指向の操作記述という側面を持っている．一般的な逐次型の手続き的プログラミング言語においては，手続き呼び出しなどのプログラムの構造は，プログラム記述時に基本的に静的に決定される．これに対して，言語処理系では実行時に文法と入力から認識される解析木に対応して意味操作の適用が行なわれるため，入力の構造に対応して動的に意味処理を行なう手続きの結合関係が与えられることになる．このため，言語処理系における計算は一般の言語のように静的に与えることが難しい．このため，言語処理系に対する計算の分割は，言語処理系記述時に手動で行なうことも可能であるが，入力プログラムの構造

に対応して動的に計算の適用が変化する言語処理系の基本的な構造から，高水準の言語による記述から自動的に行なうことが望ましい．例えば，言語の意味の形式的な記述法として発展した属性文法 [Knuth 69] では属性の評価順序をその実行モデルとして規定していない．一般的には，属性文法による記述は副作用を排除した形で行なわれることから，属性とこれを計算する関数との間の依存関係を勘案することにより，独立した計算として属性の評価を進めることができる．ただし，属性文法では中間コード列や記号表など構造を持った比較的大きなデータも，その定義と使用という面しかその評価順序の決定に影響を与えない．このため，こういったデータの処理に内在する並列性を記述することが難しいという問題がある．

本論文で示すストリームに基づいた意味処理方式では，言語処理系全体としてはパイプライン型の構成を仮定し，その上で，パイプライン型の構成で問題となる意味処理に関して，計算分割の計算方式に基づいて，ストリームによって通信する自律的な実行主体であるプロセスのネットワークを構成することによって言語処理系に並列処理を導入している．

1.2 オブジェクト指向プログラミング等との関連

論理フェーズの分割による 1 パス型の言語処理系の構成は，言語処理系構成の簡略化という点から有効であるだけでなく，部分的な処理対象の入力により処理を進めることができる点や，解析木や記号表などの中間的なデータ構造を保持しなくとも良いという利点がある．このため，属性文法に基づいたシステムでも，意味の記述に大幅な制約を加えることによって，1 パス型の処理を可能としている場合もある．マルチパス型の構成を属性文法の拡張として扱う試みも存在するが [Swierstra 91]，これは本質的な解決とは言えない．

このように 1 パス型の構成は，一般的な逐次型のコンパイラにおいては記述力の低下を意味している．このため，例えば，1 パス型の処理系において，ラベルの前方参照などを解決するにはバックパッチなど特殊な技法が要求されることになるが，これは記述のモジュール性を損なうこととなる．これに対して，手書きのコンパイラへの実現に際して，ワンパス型のコンパイラに並列プログラミングを採り入れることにより，1 パス型の言語処理系記述における問題が簡略化されることが指摘されている [Banatre 79][Andre 81]．ここでは，手書きの言語処理系を複数のプロセスに分割し，イベントによって参照と定義間の情報伝達を行なうことによって言語処理系記述を簡略化している．

後に示すように，本研究では言語処理系における意味処理を，意味処理を行なうプロセスのストリームによる通信によって行なう方式を採っている．これらのプロセスは，対応する生成規則の認識によって生成され，文法記号に対して定義されたストリームによってプロセス間が結合される．生成されたプロセスは内部状態を持ち，ストリームを介して通信を行ないながら自立的に実行を行なう．これらプロセスとストリームの

生成とそのプロセス間のストリームによる結合は，1パスの構文解析に従って行なわれる．1パス型の処理で問題となる，前方参照の扱いは一端が未結合のストリームからの入力によってモデル化され，対応するストリームにプロセスが結合され必要とする情報がストリームを介してプロセスから得られるまで自動的に実行が遅延されることにより扱われる．

以上の方式において，言語処理系の意味処理を行なうプロセスを，オブジェクト指向言語におけるオブジェクトと同様な，能動的なオブジェクトと捉えることができる．オブジェクト指向言語に代表されるオブジェクト指向の概念は，オブジェクトによる実現詳細の隠蔽，限定されたインターフェイスによる対象の抽象化，対象のモジュール化などの点で有用な概念であり，プログラム言語だけでなく，データベースシステムやシステムの解析など，広範囲に利用されるようになっている．その言語処理系への適用は，言語処理系のモジュール化や記述性の向上に有益なものである．一般に，オブジェクト指向言語では，オブジェクトはクラスと呼ばれるテンプレートに従って生成され，オブジェクトに対して直接メッセージを送ることによって処理を進める．本論文で示す方式においては，オブジェクト指向言語におけるクラスは生成規則に対するプロセスの定義に，クラスに対応して生成されるオブジェクトは生成規則の認識によって生成されたプロセスの実体に対応する．これらプロセスはオブジェクト指向言語におけるオブジェクトと同様に内部状態を持ち，ストリームをインターフェイスとして実現の詳細を隠蔽している．ただし，一般的なオブジェクト指向言語では，オブジェクトの生成はオブジェクト間のメッセージの交換によるプログラムの実行によって制御されるのに対して，本研究における方式ではオブジェクトに対応するプロセスの生成は基本的に対象言語の生成規則の認識によって行なわれ，プロセス間の通信はストリームを介して間接的に行なわれる点が異なっている．

また，多くのオブジェクト指向言語では，複数のクラス間でコードの共有を可能とする継承という概念を導入している．属性文法へオブジェクト指向の導入を行なう試みにおいても，同様に継承を実現する方式が研究されている [Koskimies 91]．これらの拡張では，生成規則によって定義される非終端記号間の解析木上での上下関係を，クラス間の継承関係とみなしている．しかしながら，文法上の継承関係と意味的なクラスの継承関係は常に一致するとは限らず，このような同一化は言語定義のモジュール性を損なうこととなる．このため，本研究ではオブジェクト指向言語における継承の概念は，言語記述のレベルには採用せず，必要に応じてプロセス間でメッセージの委譲を行なうことにより同等な機能を実現することとしている．

このような概念を採り入れることによって，従来の属性文法に基づいたシステムなどに不足していた，モジュール化された言語処理系の構成，副作用や動的な意味の扱いが，言語の生成規則に付随した意味処理の抽象化により可能となる．このような1パス型の言語処理系構成とオブジェクト (プロセス) の組み合わせは，

```
objects = context
single-pass + objects = multi-pass
```

で表されるように，1パス型の言語処理系にマルチパス型の言語処理系構成と同等な記述性を導入したものと捉えることができ [Koskimies 91]，言語処理系記述のモジュール化と記述力の向上に貢献する．

以上のように，本研究で示すストリームに基づいた意味処理方式では，プロセスによる情報隠蔽と，プロセス間のストリームによる通信ネットワークの構成によって，言語処理系構成のモジュール化と，広範囲な記述への適用を可能としている．

1.3 本論文の概要

先に述べたように，本研究では並列実行を基本とした広範囲の応用に適用可能な言語処理系記述と実行の枠組を研究することを目的としている．このため言語処理系の中心的な処理である意味処理部を並列に実行するための方式として，ストリームに基づいた意味処理の方式を提案した．ここでは，入力に対する解析木の認識により，言語処理系における意味処理を動的な実行主体であるプロセス群のストリームによって結合されたネットワークが構成され，これらのプロセス間のストリームを介した通信によって，入力に対する意味処理が実行される．この方式により，2章で示すようにコンパイラなどにおける意味処理が実現されるとともに動的に並列性の抽出が行なわれる．また，ストリーム上で後に現れる要素の値が先に現れる要素に依存して決定可能というストリームの性質によりインタープリタなどへの適用を可能としている．さらに，意味処理をプロセス単位で記述することによって，意味記述のモジュール化が実現されることとなる．

ストリームに基づいた意味処理は，対象とする言語の文法に対して，意味処理を行なうプロセスの解析木上のストリームによる結合関係を与えることによって定義される．このために実現した記述言語がSSGLである．SSGLは，YACC風のBNFに対してプロセスの解析木上の結合関係を定義する意味規則を付加したものであり，3章でその使用を記述形式の概要と簡単な記述例を示す．SSGLでは意味処理を行なうプロセス自身の定義は与えられず，プロセスの定義はSSGLの外部で与える必要がある．プロセスの記述は開発の初期にはC言語で与えていたが，プロセス記述の簡略化のため，プロセスの動的生成やストリームを介したプロセス間通信を可能とする言語SSDLを開発した．3章ではこのSSDLに関してもその記述形式の概要と簡単な記述例を示す．また，3章ではSSGLおよびSSDLの生成系の実現に関して簡単に触れる．さらに，4章ではストリームに基づいた意味処理方式の言語処理系への具体的な適用として，中間コード生成処理や記号表参照などの記述例を示す．

このストリームに基づいた意味処理方式のマルチプロセッサ・システム上での具体的な性能測定を行なうため，意味処理プロセスを実行するための実行環境を仮想マシンとして実現した．この仮想マシンは，密結合型の共有メモリ型マルチプロセッサ・システムSMiS上に，軽量プロセスの動的生成とプロセス間の通信機構などを意味処理を行なうプロセスに提供するスレッドライブラリとして実現されている．ここで，マルチプロセッサ・システムSMiSは，本研究での意味処理実行の性能評価のために設定した

ハードウェアであり，UNIX 上にシミュレータとして実現されている．これら，実行環境の実現に関しては 5 章で説明する．

本論文で示す，ストリームに基づいた意味処理方式では，入力に対応した解析木上のノードの認識に対応して，そのノードに対応する生成規則で定義されるプロセスの生成が行なわれる．6 章で示す細粒度プロセスによる実行方式は，この意味処理モデルをほぼ忠実に実現したものである．ただし，実際の適用においてはプロセスの実行順序がデータの依存関係などによって一意に決定される場合もある．このような場合に対応するため，細粒度プロセスによる実行モデルでは解析木のノードに対して定義される複数のプロセスを，同一のプロセス実行用の環境上で実行することによりプロセス間通信やプロセス管理のオーバヘッドを低減する機構を導入している．6 章では，この細粒度プロセスによる実行方式による PL/0 コンパイラの実現をもとに，コンパイラの実現技法や SMiS ハードウェア等に関する評価を，SMiS シミュレータ上でのシミュレーションに基づいて行なう．

本研究においてシミュレーションのために用いた SMiS ハードウェアでは，細粒度のプロセスの実行をサポートするハードウェア機構を仮定していない．このため，細粒度プロセスによる実行方式では並列性は得られたが，従来の逐次的な実現と比較した性能面での向上は不十分であった．このため，特殊なハードウェア機構を仮定しないで実行効率を向上する方式として，入力に対応する解析木に対してプロセスの割当を疎に行なうようにプロセス生成を抑制する，疎なプロセス割り当てによる実行方式を実現した．この方式では，プロセスの動的な統合処理を構文レベルの情報から制御することにより，プロセス生成制御のためのオーバヘッドを低減している．この方式とその評価に関しては 7 章に示す

8 章では，本論文で示すストリームに基づいた意味処理方式の，記述，応用，実現，および今後の課題に関して考察を行なう．

第 2 章

ストリームに基づいた意味処理の並列化

プログラミング言語などにおける構造は文脈自由文法によって定義することができる．しかし，文脈依存の情報に関しては文脈自由文法による構文の定義だけでは扱うことができない．このような文脈依存の情報の例としては，変数の型のような識別子に対する情報を挙げることができる．さらに，プログラムに対してコンパイラが生成するコードや，インタプリタにおけるプログラムの実行時の情報なども，このような文脈依存の情報の範疇に属していると考えることができる．本章では，このような文脈依存の情報に対する処理を並列に実行するための枠組として，意味処理を自律的な実行主体に動的に分割し，ストリームによって結合されたネットワークを構成することにより扱う，ストリームに基づいた意味処理の並列化方式を示す．

2.1 従来の方式とその問題点

コンパイラの意味解析部やインタープリタ等の言語に対する意味処理の記述を行なう場合，その言語の文法の各非終端記号の生成規則に対して意味処理を行なう動作ルーチンと呼ばれる手続きを与える方法や，各非終端記号に属性と呼ばれる値とそれらの定義関係を表す関数を定義する方法などが用いられる．ここでは，まずこれら従来の言語処理系の意味処理の方式の問題点を考える．

(1) 動作ルーチン

動作ルーチンによる意味処理は，手書きのコンパイラでも YACC[Johnson 75] のような生成系でも用いられる方法であり，構文の解析が行なわれるのに従って動作ルーチンが呼ばれ意味処理が行なわれる．ここでは，それぞれの意味処理の間の情報の受渡しは基本的に大域的な変数を用いた副作用を介して行なわれる．このような動作ルーチンに基づいた言語処理系の並列な実現としては，ソースプログラムを分割する方式など幾つかの実現がなされている [Seshadri 88][Gross 89][Koike 89].

このような言語処理系においてその意味処理を並列に行なうことを考えた場合，YACC のように大域的な変数を使用した副作用によって情報の交換を行なう方法では，言語処

理系の記述時に実行順序に依存した制御を行なう必要があることから，プログラミングの複雑さが増加し，システムの保守を難しくする原因ともなる．また，逐次的な処理を独立に適用することによる，記号表参照のアクセス制御に関する問題などが発生する[Seshadri 88]．このため，このような方法論は特殊かつ特定の言語に関してには適用可能であり，記述を行なう際に十分な注意を払えば性能面では有利となる可能性が高いが，並列な言語処理系構成のための一般的な枠組としては不十分と言わざるを得ない．

(2) 属性文法

属性文法に基づくシステムは，基本的に解析木上に定義された属性と，その値を決定するための関数を定義することによって言語処理系の記述を行なう．一般的には，属性の依存関係等に従って解析の順序が決定され，ここで決定された解析順序に従って各々の属性の値を決定するための値の評価関数が呼ばれることで意味解析処理が行なわれる[Deransart 88][Alblas 91a-b]．

属性文法による方式は，その記述の関数的な性質から上に述べたような動作ルーチンによる方式での並列実行における，並列実行制御の記述における複雑さやを排除することができ，属性の依存関係の持つ非決定性により並列性を抽出することができる[Klein 90][Kuiper 90][Jourdan 91]．しかしながら，この方式では意味処理の記述を行なう際に中間コードや記号表のような比較的大きなデータも，一般的には，１つの属性として扱う必要があることから，これらのデータの処理に内在する並列性を自然に記述することは難しい．さらに，属性文法による記述では解析木のノードに対して定義された固定個の属性の値の決定と，単一代入によるモデル上の制限がある．これは，属性文法に基づく方式でインタープリタ等における言語の動的な意味の自然な記述を難しくする原因となっている．

2.2 ストリームに基づいた意味処理

言語処理系における様々な処理は，データの並びとして表現できる場合が多くある．例えば，プログラムに対する中間コードの列や，プログラムの各部分における実行の履歴などはこの端的な例である．終端記号に対する属性値や，１回限りの計算が行なわれる列的な構造を持たない値に関しても，概念的には要素を１つのみ持つストリームとみなすことにより統一的に扱われる．ここでは，このようなデータの列をストリームとして定義し，このストリームの生成機構をプロセスとして定義する．ストリームは無限の要素を許す値の列であり，既に値の定まった要素の参照を残りの要素の生成と共に行なうことを許すようなデータ構造であり，さまざまな対象の記述に有用である[Shapiro 89][Friedman 92]．この性質により，ストリームを意味処理のためのデータ構造の基本とし，意味処理を自律的な実行主体のストリームを介した通信とすることで，動的な意味の処理や並列性の抽出を統一的な観点から扱うことを可能とすると共に，意味処理のモジュール化を行なうことが可能となる．

2.2.1 形式的定義

ストリームに基づいた意味処理方式は，基本的に属性文法の場合とほぼ同様に，次のように $SG = \{G, S, R\}$ で表される組によって定義される．ここで，G, S, R はそれぞれ次のように定義される．

(1) G は，$G = \{V_T, V_N, P, Z\}$ で与えられる基底文法である．ただし，ここで V_T,V_N はそれぞれ，終端記号および非終端記号の有限集合であり，$V_T \cap V_N = \phi$ を満たす互いに疎な集合であるものとする．また，語彙 V は $V = V_T \cup V_N$ で定義される．P は生成規則の有限集合．Z は開始記号を表し，$Z \in V_N$ である．ここで，生成規則 $p \in P$ は，$X_0 \in V_N, X_i \in V(1 \leq i \leq n)$ とすると，

$$p\text{: } X_0 \rightarrow X_1\ X_2\ \ldots\ X_n$$

で与えられる．

(2) S は意味処理を行なうプロセス間の通信に使用されるストリームの有限集合であり，記号 $X \in V$ に対して定義されるストリーム $S(X)$ は，入力ストリームの集合 $I(X)$ と，出力ストリームの集合 $O(X)$ に分割され，$S(X) = I(X) \cup O(X)$ で定義される．ただし，終端記号 $Y \in VT$ に対しては，出力ストリームは定義されず，$O(Y) = \phi$ である．これによって，全てのストリームの集合 S は，$S = \cup_{X \in V} S(X)$ によって定義される．

(3) R は各生成規則 $p \in P$ に対する意味規則の集合である．ここで，生成規則，

$$p\text{: } X_0 \rightarrow X_1\ X_2\ \ldots\ X_n$$

において記号 $X_k(0 \leq K \leq n)$ に対して定義されるストリーム $s \in S(X_k)$ を $X_k.s$ と表すと，意味規則 $R(p)$ は，$0 \leq i, j \leq n$ に対して，

$$[\ldots,X_i.a,\ldots] = f(\ldots,X_j.b,\ldots) \qquad (1)$$

または，

$$X_i.a = X_j.b \qquad (2)$$

で定義される．ここで，$a \in O(X_i)$, $b \in I(X_j)$ である．上の意味規則で，(1) は f で表されるプロセスが右辺に現れるストリームから入力を行ない，左辺に現れるストリームへの出力を行なうことを，(2) はストリーム $X_i.a$ からの入力をストリーム $X_j.b$ へコピーし出力することを表す．

2.2.2 基本的な実行方式

ストリームに基づいた意味処理方式においては，入力に対して文法 G に従って構文解析を行なった結果一意な解析木を得，意味規則に従ってストリーム上のデータの値を決定してゆくことによって基本的な実行が行なわれる．すなわち，言語処理系の定義時に，意味解析プロセスは各生成規則に対して定義される．各意味解析プロセスには，対

応する入力ストリームおよび出力ストリームが複数存在し，生成規則に対して文法記号を介したストリームとプロセスの結合関係が与えられる．これによって，解析木上に定義されたプロセスのストリームによる結合関係が定義される．

言語処理系に対して入力が与えられると，定義された文法に従って構文解析が行なわれた結果，解析木の認識が行なわれる．ここで認識された構文木に対して，言語処理系定義時に与えられた生成規則に対するプロセスのストリームによる結合関係を適用することによって，解析木に対応したプロセスの結合ネットワークが構成される．

概念的には，プロセスの生成処理は，構文解析部で対応する生成規則の認識が行なわれると同時に行なわれ，生成されたプロセスは生成と同時に実行を開始すると考えることができる．解析木の特定のノードの認識に対応して生成されたプロセスは対応する解析木上の親，兄弟，子のノードに属したプロセスの出力する複数のストリームを入力として取り，これを用いて出力ストリームの値を計算する．プロセスはストリームからの入力およびストリームへの出力に対する全データの計算が終了した時点でその実行を終了し消滅する．

このモデルに対する基本的な実行機構としては，解析木を構築しその上で解析木の走査を行ないながらプロセスの実行を行なう方式や，ストリームに対するプロセスからの値の送受信による変化の伝搬によりコルーチン的にプロセスを切り替えながら実行を進める方式などが考えられる．現在の実現は基本的に後者の方式によっている．この実現の詳細に関しては6章および7章で説明する．

ストリームに対する変化の伝搬による実現をとった場合，プロセスの生成とストリームによる結合の処理が構文解析と同時に可能な場合，意味解析の実行時には解析木の構造は不要である．このため，構文解析部と意味解析部は同時に動作させることが可能であり，必ずしも実際の解析木を作る必要はない．言い替えると，この場合には，意味の実行に必要な解析木上の構造は，動的な実行主体であるプロセスのストリームによる結合ネットワークに変換され保存されていることとなる．

ここで導入したストリームに基づいた意味処理方式は，ストリームを属性文法における属性と，プロセスを属性の値を決定する関数と対応させると，属性文法を拡張したものと考えることもできる．すなわち，属性文法での合成属性は解析木の下から上に値が受け渡されるストリーム，相続属性は上から下に値が受け渡されるストリームに対応し，これらの属性の値の決定は本方式ではプロセスによって行なわれることとなる．一般に，属性文法における属性の評価では，ある属性の値の決定によって他の属性の値が決定されるという方式をとっており，ある属性が自身の値に依存するような定義を許さない場合が多い．本方式では，ストリームのある要素が，自身のすでに決定された要素の値に依存して，その値を決定することを許している．同様に，属性文法における属性の決定は副作用を持たない関数によって行なわれることを想定している場合が多いが，これは入出力や表のような破壊的な操作を必要とするデータ構造の扱いの導入を難しくしている．これに対して本方式では，それまでのプロセスの通信の履歴をプロセス内に保持することが可能であり，これによりプロセス内に副作用の影響を閉じ込めることを可能としている．以上のように，本方式では様々な点で属性文法に基づいた一般的なシ

ステムに対して，記述可能な対象が拡張されている．

2.3 コンパイラにおける意味処理の概要

本論文で示す，ストリームに基づいた意味処理では，言語処理系における意味処理を，ストリームによって結合された自律的な実行主体であるプロセスのネットワークによって構成する．ネットワークを構成する各プロセスは，ストリームを介して通信を行なうことにより意味処理を行なう．このネットワークの構築を行なうための規則は，記述対象の言語等に対して，自律的に意味処理の実行を行なう実体であるプロセスと，プロセス間の通信路であるストリームによる解析木上でのプロセスの結合とを，その生成規則に対して指定することによって与える．意味処理を行なうプロセスは，生成規則に対応する入力の構文の認識に対応して生成され，解析木のノードを介して結合されたプロセスとストリームを介した通信を行なうことによって意味処理を行なう．意味解析処理を行なう動的な実行主体であるプロセスは，これに対して定義されたストリームに対して必要な入出力操作を終えた時点で実行を終え消滅する．以下，本節ではコンパイラにおける典型的な意味処理を例にとり，そのストリームに基づいた意味処理方式での扱いを示す．

(1) ストリームによる中間コード生成処理

ここでは，言語処理系における並列な意味処理の具体的な例として，まず，コンパイラにおける中間コードの生成処理を考える．一般に，中間コード列は概念的には解析木の下から上に順に決定され，中間ノードでボトムアップに合成されながらルートノードに集められる．ストリームに基づいた意味処理方式では，解析木の各ノードに対してそのノードをルートとする部分木に対応した中間コード列をストリームに出力するプロセスを設けることによって図 2.1 に示すように，中間コードの合成処理が並列に実行される．この図において，破線は while 文に対する部分的な解析木の構造を表し，while 文の条件式の中間コードの処理を行なうプロセス，while 文のボディーの中間コードの処理を行なうプロセス，while 文自身の中間コードの処理を行なうプロセスはそれぞれ白抜きの丸で表現している．また，これらのプロセスを接続するストリームに関しては，データの転送方向を表す矢印で示している．

この例で，入力プログラムに対応する中間コード生成に対応するプロセスのネットワークは，解析木の構造に沿って，各ノードにコードの合成を行なうプロセスを持ち，これらが上昇方向に中間コードの出力を行なうようにストリームで結合された木構造として構成される．各ノードにおいて中間コードの合成を司るプロセスは，下位ノードに対応した中間コードを，これを合成するプロセスから受けとり，上位ノードに対して合成されたコード列を出力する．このように中間コードの合成処理を行なうプロセスから構成された木構造の下から上に向かって中間コードの生成と消費を同時に行なうことによって，パイプライン的な中間コード生成処理が実現される．ここでの部分解析木に対応する中間コード列の決定処理には，基本的に大域的な依存関係がなく，部分的な解析

木に対して局所的に独立に処理を適用することが可能である．これによって，構文解析動作とともに，コード生成を行なうプロセスのネットワークを動的に構築しながら処理を進行することができる．

(2) ストリームによる記号表参照管理

次に，言語処理系において記号表の参照を並列に行なう場合を考える．ここで，言語処理系における記号表の実現方式としては幾つかの方式が考えられるが，多く用いられているのは次の２種類の方式である．１つは，多くのコンパイラで採られているように，集中的に管理される大域的なデータ構造を１つ設け，これに対して複数の場所から，更新および参照を行なう方式である．この方式では，記号表に登録されたデータの更新と参照が矛盾を起こさないよう言語処理系の記述を行なう際に何らかの方式を用いて排他制御を行なう必要がある．また，全ての参照が同一のデータに集中する結果となるため，記号表の並列な実現としては有効ではない．もう１つは，属性文法による記述に見られるように，宣言部において有効な記号の情報を収集し，各ノードにおいて有効な環境に関する情報をコピーする方式である，この方式では各地点において有効な環境をコピーする必要がある．また，エラーの発生時に記号表内の情報を書き換えるという技法を用いることも難しい．このような，記号表を構成するデータの分散と，データの更新など集中的な管理を両立させるには，集中的に管理されていた記号表を論理的に意味を持つ単位に分割することが有効である．このような記述によって，ストリームに基づいた意味処理では，記号表を管理するプロセスへのアクセスを緩和させると共に，集中型の実現での記号表データの書き換えを両立させることができる．

具体的には，各々の論理的な宣言に対応して記号表の管理を行なうプロセスを設ける．このプロセスは，論理的な各宣言部に対して，対応する入力構造が認識されることによって生成される．この局所的な記号表管理を司るプロセスと，変数の参照ノードなど局所的な名前に対する記号表の参照を行なうプロセス間の通信によって記号表の参照を実現する．ここでは，参照側のプロセスと，記号表管理を行なう被参照側のプロセス間で，参照したい名前と，参照結果に対応する記号の情報を相互に伝達する必要がある．この参照の制御は，参照側のプロセスと被参照側のプロセスとの間で双方向のストリームの組を張ることにより実現される．ここで，参照側のプロセスと被参照側のプロセス間のストリームの組は，記号表管理を行なうプロセスへの参照要求の送信と結果の返送に利用され，これによって複数の参照に関する排他制御が自動的に行なわれる．

すなわち，図 2.2 に示すように，記号表参照を行なうプロセスでは，記号表管理プロセスに対して識別子に対応した記号表の参照要求を送り，記号表の参照を行なった結果である記号に関する情報をもう一方のストリームから得る．一方，記号表管理プロセスでは，参照プロセスに接続されたストリームから受けっ取った参照要求に対する識別子の参照結果を，対となるストリームを介して参照側のプロセスに送付する．なお，図 2.2 ではブロックに対する部分的な解析木の構造を破線で表している．記号表を管理するプロセスと，名前を参照するプロセスそれぞれについては白抜きの丸で表現し，これ

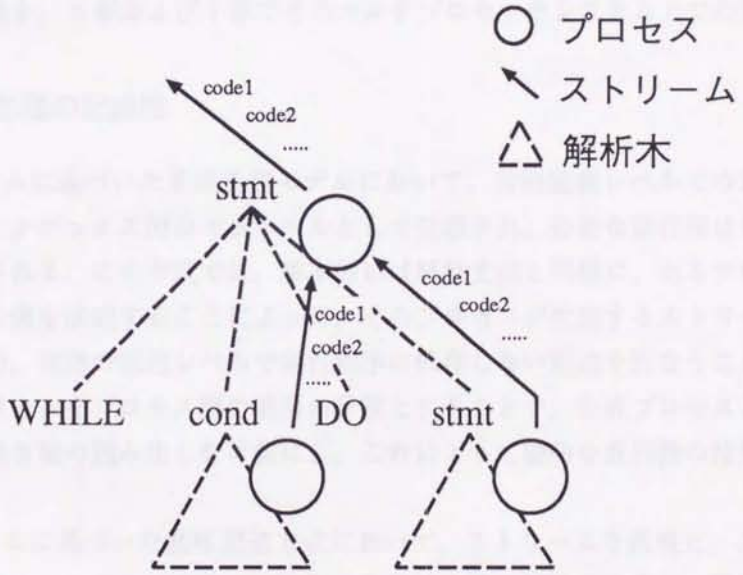


図 2.1 パイプライン型のコード生成

らのプロセスを接続するストリームに関しては，データの転送方向を表す矢印で示している．

一般的に，言語のスコープ規則は木構造を構成するように設計されていることが多い．これに従って，記号表参照を実現するためのプロセスのネットワークも，このスコープ規則に従った木構造を構成する．ここで，リーフノードは識別子に対して記号表参照を行なうプロセスに対応し，中間ノードはそれぞれ論理的な宣言部に対応する記号表を管理するプロセスに対応する．親子関係を構成するプロセス間は，識別子参照要求を送信するための上昇方向のストリームと，参照結果を返すための下降方向のストリームの対によって結合される．言語のスコープは，ここでは中間的な記号表管理プロセスの階層に変換されることとなる．

以上で述べたような，言語処理系における並列な意味処理に関しては，4章でその具体的な記述を，6章および7章でそのマルチプロセッサシステム上での実現を示す．

2.4 意味処理の記述性

ストリームに基づいた意味処理モデルにおいて，言語定義レベルでの意味解析プロセスはブラックボックス的なモジュールとして定義され，必要な副作用はそのプロセス内部に吸収される．この方式では，基本的には属性文法と同様に，あるプロセスの入力ストリームの値を決定することによって，そのプロセスが生成するストリームの値が決定されるため，言語の記述レベルで実行順序に依存しない記述を行なうことができる．また，ストリームをプロセス間の通信の手段とすることで，生成プロセスと消費プロセス間で部分的な値の読み出しを可能にし，これによって動的な並列性の抽出を容易としている．

ストリームに基づいた意味記述方式において，ストリームを属性と，ストリームを介して受け渡される値をリストにしたものを属性の値と対応させると，属性文法を並列実行の機構とストリームの導入により拡張したものとみなすこともできる．ここでは，ストリームの導入によって従来の属性文法によるシステムに比べ処理可能な対象が広がり，インタプリタ等における動的な意味の記述を行なうことが可能になっている．例えば，ストリームに基づいた意味記述の枠組で解析可能なクラスは，全てのストリームの要素を1つだけとし，ストリームを属性文法での属性と対応させると，これで記述可能な対象は非循環な属性文法で記述可能な対象に対応する．これに加えて，ストリームに基づいた意味記述方式では，同一のストリームにおける複数の要素間で依存関係を持った記述が可能なことから，実際には，ストリームの要素の依存関係にループがなく，ストリームの要素が，同一のストリーム上でその要素以降に現れる要素に依存しない対象を記述可能としている．

また，近年，属性文法にオブジェクト指向の概念を採り入れることにより言語処理系記述のモジュール化を可能とする試みが行なわれている．これらオブジェクト指向属性文法 [Shinoda 90][Koskimies 91] と呼ばれる枠組は，未だ統一された概念ではない

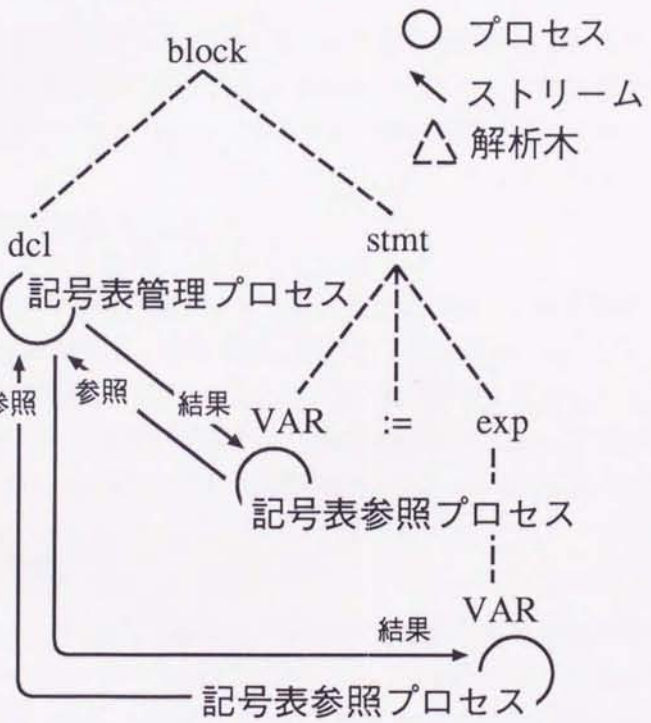


図 2.2 ストリームによる記号表参照の実現

が，基本的には属性文法とオブジェクト指向の概念を融合することにより，言語記述のモジュール化や，動的な意味の記述を可能とすることを目的としている．ストリームに基づいた意味記述モデルにおいても，同様の概念を自律的な実行主体であるプロセスをオブジェクトと捉えることにより，プロセスによる情報のカプセル化により記述のモジュール化や動的意味の記述を実現することができる．例えば，ストリームに基づいた意味記述方式では，定義対象の言語の具象構文に対して変更が行なわれた場合でも，具象構文上のストリームの結合関係を変更するだけで対応することができる．また，様々な言語コンストラクトに対応する処理を行なうプロセスをライブラリ化することにより，言語処理系のプロトタイピングなどに適用することも可能となる．

第 3 章

記述言語

本章では 2 章で示したストリームに基づいた意味処理に関して，構文に対してプロセス単位で意味の具体的に記述する言語である SSGL と，意味処理を行なうプロセスの動的な生成と通信のアルゴリズムを記述するための言語 SSDL について説明する．

3.1 意味処理記述言語 SSGL

2 章で述べたストリームに基づいた意味処理の記述は，YACC[Johnson 75] に類似した BNF による文法定義に，意味規則と呼ばれるプロセスとストリームの結合関係の定義を行なうための規則を付帯させた，SSGL と呼ばれる言語によって与えられる．ここでは，この SSGL の記述形式に関して説明を行ない，続いて SSGL による簡単な記述例を示す．なお，SSGL の文法の定義に関しては appendix-A に示す．

3.1.1 記述形式の概要

SSGL の定義は，YACC の場合と同様に，宣言部と文法および意味規則の定義部を %% で区切って記述する．YACC の場合には，定義ファイルの最後に C 言語の定義を置くことを許しているが，SSGL では，宣言および文法記述以外の記述は他のモジュールで行なう．なお，SSGL 記述中では '#' 以降はコメントとして扱われる．

(1) 構文および意味規則の定義

SSGL における文法および意味規則の定義は，YACC 風の BNF による生成規則の定義の後に意味規則を付加することで行なう．文法の定義は YACC と同様に，定義する非終端記号と生成規則を ':' で区切って指定し，';' で定義の終りを表す．同一の非終端記号に対して複数の生成規則を定義する場合は，'|' によって生成規則を区切って指定する．YACC の記法では意味規則は生成規則中の任意の位置に複数挿入することが可能であるが，SSGL では各生成規則に対して高々 1 つの意味規則を与える．この意味規則は各生成規則の後に付加することで定義する．

SSGL における意味規則は，YACC の場合と同様に '{', '}' で括って指定し，ストリームからストリームへのコピー，プロセス生成とその解析木上でのストリームによる

結合関係の記述を与える．意味規則は '=' によって区切られた右辺のストリームまたはプロセスから，左辺のストリームへの出力を表す．あるプロセスに対して出力ストリームが複数ある場合，出力ストリームを ',' で区切り，それらを '[',']' でくくることによって指定する．プロセスに対して出力ストリームが存在しない場合には，'=' から左は省略する．また，SSGL ではプロセス間での双方向通信の記述の簡略化のため，アトミック・ストリームと呼ばれるストリームを用意している．アトミック・ストリームは概念的には双方向のストリームの対として扱われるとともに，3.2 節で説明するようなアトミックな送受信を行なうプリミティブによって多対 1 プロセス間の排他的な通信に使用される．このため，アトミック・ストリームに対しては '=' による意味規則の定義は，右辺のストリームあるいはプロセスから左辺のストリームへの出力を表すと共に，左辺のストリームから右辺のストリームへの出力を同時に表している．

この意味規則中での各記号に対するストリームの参照は，生成規則中に現れる記号名の後に，ストリーム名を '.' で区切って指定することにより行なう．ある生成規則中に同一の記号が現れる場合には，それぞれの記号に対するストリーム参照を区別するため，記号名の後に生成規則中でのその出現順序を '[',']' でくくって指定する．なお，ここでの記号の出現順序は 0 から始めるものとする．すなわち，ある記号 X に対するストリーム S の参照は，その記号の生成規則中における出現順序を $n(\geqq 0)$ とすると，X[n].S と表現する．この時，n=0 ならば，X.S と省略することが可能である．

意味規則中のプロセスは，プロセス名の後にプロセスの入力引数を '(',')' で括って指定する．プロセスの入力引数はストリーム，定数，関数のいずれかである．プロセスは必要に応じて入力ストリームから入力を行ない，計算した結果を '=' の左辺に指定された出力ストリームに送付する．ここで，プロセスの引数として与える定数は数値定数，文字列定数，文字定数のいずれかでありこれらの表記は C 言語の表記に従う．これ以外の識別子に関しては，SSGL 記述から C 言語への変換後に展開される記号定数として扱われる．プロセスの入力引数に現れる関数には，C 言語によって記述された関数を指定する．また，プロセスの実体は SSGL 自身では記述せず，C 言語または次節で述べる SSDL 言語によって記述する．意味規則における各プロセスはその定義の前に SEQ を前置することによって逐次化の指定を行なうことができる．逐次化指定を行なったプロセスは C 言語で定義された関数として扱われ，5 章および 6 章で説明したような方式により，プロセスの統合処理が行なわれる．

SSGL 記述において，終端記号に対する出力ストリームや，5 章および 6 章で述べるような逐次化対象のプロセスからの出力は，順序を持った値の列ではなく単なる値として扱われる．このため，概念的にはこれらの値に対してストリームという名称を用いるのは正確ではないが，ここでは簡単のためこれらを含めてストリームという名称を使用し，区別の必要がある場合は，その都度説明を加えることとする．

(2) 宣言

SSGL の宣言は，

・ストリームに対する型の宣言

・SSGL 記述から生成されたファイルへの他のファイルの挿入の宣言
・終端記号および区切り文字の宣言
・演算子の結合規則の宣言

からなる．これらの宣言は，YACC と同様に '%' で始まるディレクティブによって与える．

・型宣言

ストリームに対する型宣言は，アトミック・ストリーム型のストリームと，終端記号の属性値，および逐次的に評価されるプロセスの出力ストリームの型を宣言する．宣言を与えられないストリームについては，通常のストリーム型として扱われる．ここで，アトミック・ストリーム型のストリームは，%atomic の後にストリーム名を並べることで宣言する．また，非終端記号の属性値および逐次的に評価されるプロセスの出力ストリームの型については，%type の後にストリームの型名とストリーム名を指定することで宣言する．ここで与える型名は，変換後の C 言語プログラム中で使用可能な C 言語の型名であり，以下に述べる %include によって指定する C 言語のファイル中で，typedef または，#define を用いて定義する．

・ファイル挿入宣言

ファイル挿入は，%include の後にファイルのパス名を文字列形式で与えることで宣言する．この扱いは C 言語のプリプロセッサにおける #include ディレクティブと同様である．

・終端記号宣言

SSGL 記述は変換系によって YACC 記述へと変換されるため，生成されたプログラムにおける構文解析系と字句解析系のインターフェイスは YACC と同一となっている．つまり，構文解析系では，yylex という手続きを呼びだしその返値をトークンの識別値とする．トークンの属性値は yylval という名前の変数を介して受渡しが行なわれ，これが終端記号に対応するストリームの値となる．

SSGL 記述では，YACC の記述と同様に字句解析系を構文解析系と別のファイル中に記述することも可能である．この場合は，YACC の場合と同様に，ある文法記号が終端記号であることを，%token の後に文法記号名を列挙することで宣言する．SSGL 記述内で字句レベルの宣言を行なうことも可能であり，この場合には，使用する終端記号および，区切り文字列の宣言を与える必要がある．この終端記号の宣言は，%token の後に終端記号名を指定し，その後に終端記号の字句レベルの表現を表す正規表現を文字列の形で ':' で区切って指定する．区切り文字列の宣言は，%skip の後に文字列形式でその正規表現を指定する．これら終端記号および区切り文字列に対しては，'{','}' で区切って意味規則を指定することができる．意味規則中には，C 言語の関数呼び出し，または終端記号に対するストリームの値の定義を記述する．この意味規則中で終端

記号または区切り文字列の字句レベルの表現が必要な場合，'@' によってこれを参照することができる．終端記号の意味規則中では，終端記号の後にストリーム名を '.' で区切って指定することにより終端記号のストリームを参照することができる．

・オペレータ宣言

オペレータ宣言は，YACC の宣言と同様に %nonassoc, %left, %right の後に終端記号名を並べることにより行なう．%nonassoc, %left, %right はそれぞれ，終端記号で指定された演算子が非結合，左結合，右結合であることを表すもので，先に宣言されたものほど優先順位が低いものとして扱う．

3.1.2 記述例

ここでは，SSGL による意味記述の例として，入力とした与えられた数値のリストをソートする意味処理の記述を示す．以下，val は数に対応する終端記号 NUM の整数値を表し，out は非終端記号に対応する数値のリストをソートした結果に対応するストリームである．以下の定義で，output は入力ストリームから送られてくるソートされた列の値を表示するプロセス．lsort は引数として 2 つのソートされた数値からなるストリームを受けとり，これをマージしたストリームを生成するプロセス，sort はソートされたストリームと，数値を引数とし，これらをソートした数値列を生成するプロセスである．ただし，SSGL 定義の範囲内ではプロセスの実体を定義することはできないので，これらのプロセスの定義はここでは与えない．以下の定義で，1 ～ 8 行までは宣言部，10 ～ 23 行は構文と意味規則の定義に対応している．宣言部において，1 ～ 2 行は生成後の C プログラムへのヘッダ・ファイルの挿入の指定，4 行目は val に対する型宣言，6 行目は入力に対する区切り文字列の正規表現の定義，7 ～ 8 行目は，数に対応する終端記号 NUM の正規表現と，数値への変換を行なう意味規則の定義である．10 ～ 23 行の文法と意味の定義部では，sort, lists, list の 3 つの非終端記号を定義している．ここで，sort は入力全体に対応する非終端記号であり lists と等価なものとして定義され，この認識によってソートされた数値列の出力を行なうプロセス output の生成を行なう．lists は '[',']' で区切られたリスト列の構文に対応する非終端記号であり，プロセス lsort を生成することにより 2 つのストリームから送られてくるソートされた数値列をマージすることによりソートし出力する．list は数値列に対応する非終端記号であり，プロセス sort を生成することによって第 2 引数で与えられる数値を入力ストリームとして与えられるソートされた数値列中に挿入することによりソートを行なう．

```
1  %include        "stdio.h"
2  %include        "process.h"
3
4  %type int val
5
6  %skip             "[ \t\n]+"
7  %token NUM      : "[0-9]+"
8                        { NUM.val = atoi(©); }
```

```
 9  %%
10  sort
11          : lists         { output( lists.out); }
12          ;
13  lists
14          : lists '[' list ']'
15                          { lists[0].out = lsort( lists[1].out,
16                                                  list.out); }
17          |               { lists.out = NULL; }
18          ;
19  list
20          : list NUM      { list[0].out = sort( list[1].out,
21                                                NUM.val); }
22          | NUM           { list.out = sort( NULL, NUM.val); }
23          ;
24  %%
```

この記述に対して，入力として [4 2] [3 7] を与えた場合の解析木，プロセスのストリームによる結合の様子とストリームを流れる値を図 3.1 に示す．この図において，破線は入力に対して生成される解析木の構造を表している．また，output，lsort，sort などに対応するプロセスはそれぞれ白抜きの丸で表現し，これらのプロセスを接続するストリームに関しては，データの転送方向を表す矢印で示し，各ストリームを受け渡される数値列を '[',']' で括って表している．図に示すように，list 及び lists に対応するノードに対して数値列のソートを行なうプロセスが入力の構造に従って配置され，これらがボトムアップにソートされた数値列がストリームによって受け渡すことにより，パイプライン的にソーティング処理が行なわれ，ソートされた数値列が入力全体に対応する非終端記号 sort に対して生成されたプロセス output で受理される．具体的には，入力されるリスト中の数値 4, 2, 3, 7 それぞれに対してプロセス sort が生成され，各リスト [4 2] と [3 7] に対してプロセス lsort が生成され，入力全体に対応してプロセス output が生成される．これにより，計 7 個のプロセスが生成され，ソーティングを行なう．

3.2 プロセス記述言語 SSDL

SSGL は文法とプロセスの結合関係を記述するが，実際の意味解析処理を行なうプロセスの定義に関しては，SSGL による意味記述とは別に，外部から与える必要がある．初期の段階では，プロセス生成やストリーム操作を行なうライブラリを用いて C 言語によりプロセスを記述していた．しかしながら，これらの記述は可読性が低く，特殊なプロセス実現方式などの適用を行なうことが繁雑であるという問題点があった．このため，ストリームに基づいた意味処理の高水準な記述を可能とし，様々なプロセス実現技法の適用を可能とするために，ストリーム操作を行なうプロセス定義を高位なレベルで与えることを可能とする言語を用意した．この言語は SSDL と呼ばれ，C 言語風の制御構文に加えて，プロセスの実体定義，プロセスの動的生成，ストリームを介した通信機構，C 言語によって記述された手続きの呼びだしを行なうための機構などを備

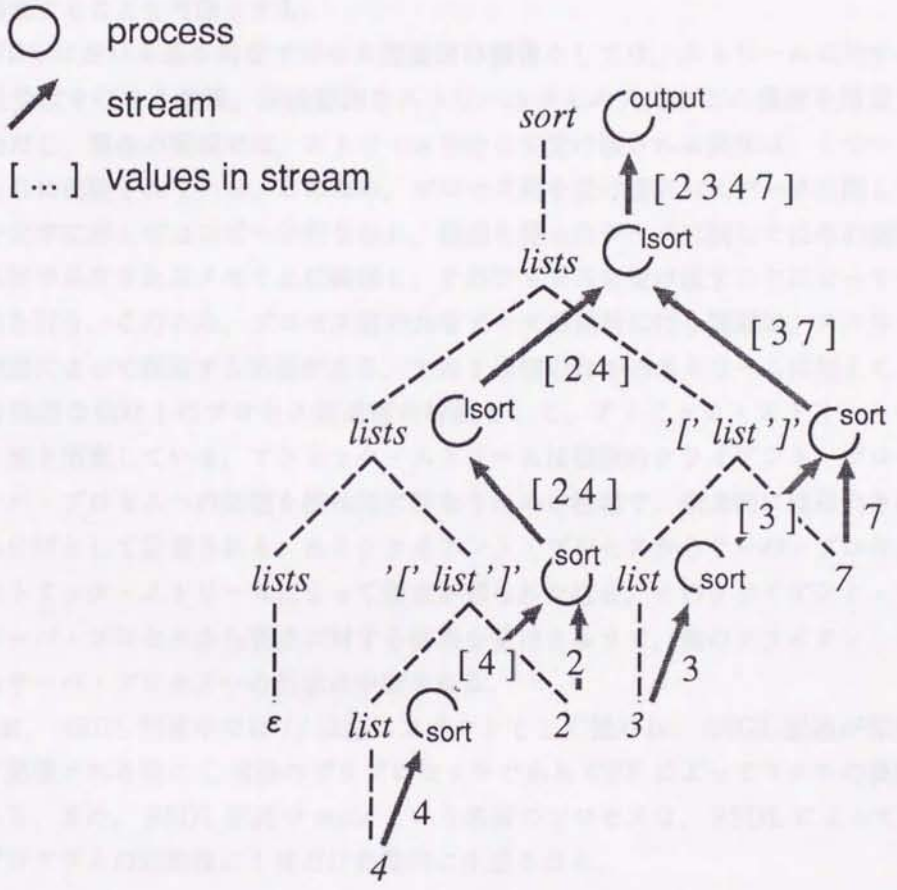


図 3.1 ソーティングの実行の様子

えている．

3.2.1 記述形式の概要

SSDLは，基本的にはCSP型のメッセージ通信モデル[Hoare 89]によるプロセス記述のための言語である．ただし，CSPやCSPから派生したOccam[Inmos 88]などが動的なプロセス生成を許さない静的な言語であるのに対して，SSDLは動的なプロセス生成のための機構を持っている．このような動的なプロセス生成の機構は，SSGLによる定義と入力の解析によって与えられた意味解析プロセスのネットワークを，動的に再構成することを可能とする．

SSDLにおける基本的なプロセス間通信の機構としては，ストリームに対するデータの送受信を行なう機構，非決定的なストリームからの入力などの機構を用意している．ただし，現在の実現では，ストリームを介して受け渡される実体は，1ワードに収まるものに制限されている．このため，プロセス間を受け渡されるデータに関しては，整数や文字に関してはコピーが行なわれ，構造を持ったデータに関してはその実体をプロセス間で共有されるメモリ上に確保し，そのアドレスを受け渡すことによってデータの共有を行う．このため，プロセス間の共有データの更新に伴う問題は，ストリームによる同期によって保証する必要がある．1対1通信のためのストリームに加えて，SSDLでは排他的な他対1のプロセス間通信の枠組として，アトミック・ストリームというデータ型を用意している．アトミック・ストリームは複数のクライアント・プロセスからサーバ・プロセスへの通信を排他的に行なうための機構で，概念的には逆向きのストリームの対として定義される．あるクライアント・プロセスからサーバ・プロセスに対してアトミック・ストリームによって要求が送られた場合，そのクライアント・プロセスがサーバ・プロセスから要求に対する結果を受けとるまで，他のクライアント・プロセスのサーバ・プロセスへの要求は中断される．

なお，SSDL記述中では//以降はコメントとして扱われ，SSGL記述が変換系によって処理される前にC言語のプリプロセッサであるCPPによってマクロの展開が行なわれる．また，SSDL記述中mainという名前のプロセスは，SSDLによって記述されたプログラムの起動後に1度だけ自動的に生成される．

(1) 宣言

SSDLの記述は定数定義，型宣言，インポート宣言，プロセス定義からなる．定数定義は，数値，文字列，文字定数に対して名前付けを行なうものである．各定数の表記はC言語のものに従う．インポート宣言は，SSDL記述から生成されたC言語のプログラム中に指定したファイルを取り込むための指定である．型宣言は，SSDL記述外で宣言された型(外部型)をSSDL内で使用するための宣言である．SSDLでは，基本型として整数型(int)，文字型(char)，ストリーム型(stream)，アトミック・ストリーム型(atomic)のみを提供している．これ以外の型は，C言語により別ファイル中に記述し，importにより取り込むことにより使用する．SSDL中ではこれらの型を直接扱

うことはできないが，C 言語の手続きを呼び出すことにより，外部型の操作を行なうことが可能である．なお，SSDL から C 言語への変換後は，外部型，ストリーム型，およびアトミック・ストリーム型の変数は，それぞれ対応する C 言語の型のポインタ型として扱われる．

(2) プロセス定義

SSDL におけるプロセスの定義の構文は，ほぼ ANSI スタイルの C 言語の関数の宣言スタイルを踏襲しており，プロセスの呼びだし形式を宣言するヘッダと，プロセス本体に対応するブロックから構成される．プロセスヘッダは，定義するプロセスの出力ストリームの後に '<–' を置き，その後にプロセス名，入力引数を '(', ')' で括ることにより指定する．プロセスに出力ストリームが存在しない場合には '<–' 以前は省略する．出力ストリームが複数存在する場合には，出力ストリームを '[', ']' で括って指定する．プロセスの入力引数は，変数またはストリームである．変数は C 言語と同じく型名の後に変数名を列挙することで宣言する．ストリームの宣言については次項で説明する．プロセス本体に対応するブロックは，局所変数の宣言とステートメントから構成され，これらを '{','}' でくくって指定する．SSDL における局所変数の宣言は，C 言語と同じスタイルをとり，型名の後に変数名を ',' で区切って指定する．

(3) ストリーム

SSDL におけるストリームには通常のストリームと，アトミック・ストリームの 2 種類が用意されている．ストリームおよびアトミック・ストリームの型名はそれぞれ Stream および Atomic であるが，プロセス定義のヘッダ中では，ヘッダ宣言の簡略化のため，型名の指定をしない変数を通常のストリーム型の引数，名前の前に '@' を付加したものをアトミック・ストリーム型の引数として扱う．

(4) ステートメント

SSDL の制御構文の構成はほぼ C 言語と同様である．異なる点は switch，goto など幾つかの構文が除かれた点と，プロセスの動的な生成を行なう構文 (process ステートメント) と，複数のストリームからの入力待ちを行なう構文 (select ステートメント) が追加された点にある．その他の制御構文の扱いは C 言語と同一であり，条件式の評価も C 言語と同じく，値が 0 の場合を偽，それ以外を真として扱う．

(5) select ステートメント

select ステートメントは複数のストリームからの入力を非同期的に行なうための制御構文であり，select の後に，複数のストリームの選択文を '{','}' で指定する．ストリームの選択文は，

```
ストリーム -> 変数
        ステートメント
```

または，

アトミック・ストリーム --> 変数
ステートメント

という構文で指定する．selectステートメントの実行時には，各ストリームの選択文で指定されたストリームが入力可能かどうかが順にテストされ，入力可能なストリームが存在した場合，対象となるストリームからの入力が，ストリーム選択文で指定した変数に行なわれ，対応するステートメントが実行される．各ストリーム選択文は，

: ステートメント

を後ろに付加することによりストリームが閉じられている場合に実行されるステートメントを指定することができる．また，ストリーム選択文の前に，

(式) &&

を前置することにより，特定のストリーム選択文によって指定されたストリームを監視するかどうかを制御することができる．この場合，指定した式の値が真であればストリームの監視を行ない，偽であればストリームの監視を行なわず，そのストリーム選択文を無視する．

(6) processステートメント

processステートメントはプロセスの動的な生成を行なうための構文である．プロセスステートメントは，processの後にプロセスの生成文を指定する．複数のプロセスを同時に生成したい場合は，プロセスの生成文を'{','}'でくくって列挙する．プロセスの生成文は，'<-'の左辺に出力ストリームを指定し，右辺にはプロセス名の後に入力引数を括弧で括って指定する．プロセスの生成文で指定する入力引数はストリーム，変数，または定数を指定する．プロセスヘッダの定義と同様に，プロセスに出力ストリームが存在しない場合には，'<-'から左は指定する必要がない．また，出力ストリームが複数ある場合には，'[',']'でくくって指定する．

(7) 式

SSDLの式もステートメントの場合と同様に，C言語の構文をほぼ踏襲している．主な違いは，利用頻度の低い幾つかの演算子と配列や構造体の参照構文を削除し，ストリームへの送受信を行なう構文を追加した点にある．なお，式中に現れる関数呼びだしは，対応するC言語の関数呼び出しに変換される．

(8) ストリーム入出力式

ストリーム入出力式は，ストリームに対する出力，ストリームからの入力，ストリーム中にデータが存在するかどうかをテストするための式である．

ストリームへの出力は，

ストリーム <- 式

という構文を使用する．この場合，式の値が左辺のストリームへと出力される．ストリームへの出力構文には，ストリームへの値の出力を行なうと共にストリームを閉じる操作も用意されている．この構文では，'<-' の右辺に ',' で区切った式を '[',']' でくくることにより，式の値を順に左辺のストリームへ出力した後にストリームを閉じることができる．'[',']' 中の式を省略すると，単にストリームを閉じる操作となる．SSDL ではストリームは有限長のバッファとして実現されることを想定しているため，Occam や CSP のように入力プロセスと出力プロセスの間で同期は行なわないが，出力対象のストリームのバッファがいっぱいの場合には対応するストリームの入力操作が行なわれるまで出力操作は中断されることとなる．これらストリームへの出力式は値を取らない式である．SSDL でのストリームからの入力には，

ストリーム -> 変数

という構文を使用する．ストリーム中にデータが存在すればデータが取り出され変数に代入される．データが存在しない場合には，他のプロセスによってストリームへのデータの出力が行なわれるか，ストリームが閉じられるまで入力操作は中断される．SSDL では，プロセス間でストリーム自身をストリームを介して受け渡すことが可能であり，ストリームへのストリームの出力構文には上で述べたものと同一の構文を用いるが，受け手側で受けとったデータをストリームか，他のデータかを区別できるように，特殊な構文を用意している．これは，ストリーム入力構文の後にストリーム型の変数を '<','>' で括って指定する．入力データがストリームであった場合，指定したストリーム変数に入力したストリームが代入され，その他の場合，ストリーム変数の値は NULL となる．ストリームからの入力式の値は，ストリームから入力が行なわれた場合は真，その他の場合，つまりストリームが閉じられデータが存在しない場合は偽となる．ストリームからの入力を伴わないで，ストリームが閉じられておりその中にデータが存在していないかどうかを確かめる構文は，ストリームからの入力構文と同様の構文で，入力変数の代わりに '[]' を指定する．ストリームが閉じられ，ストリーム中にデータが存在しなければ式の値は真，それ以外の場合偽となる．

(9) アトミック・ストリーム入出力式

SSDL では，排他的な他対 1 のプロセス間通信を行なうための枠組として，アトミック・ストリームというデータ型を用意している．アトミック・ストリームは複数のクライアント・プロセスからサーバ・プロセスへの通信を排他的に行なうための機構である．あるクライアント・プロセスからサーバ・プロセスに対してアトミック・ストリームによって要求が送られた場合，そのクライアント・プロセスがサーバ・プロセスから要求に対する結果を受けとるまで，他のクライアント・プロセスのサーバ・プロセスへの送信は中断される．

クライアント・プロセスから，アトミック・ストリームに対する要求の送信は，結果の受信とを対にして，

ストリーム <-- 式 --> 変数

という構文で行なう．ここでは，まず，サーバに対する要求を表す式の値がストリームを通してサーバ・プロセスに対して送られ，サーバ・プロセスからの結果が変数に返される．要求が送られると，結果が返されるまでプロセスの動作は中断する．サーバ側では，

ストリーム --> 変数

という構文によって，クライアントからの要求を受けとる．要求に対する結果の返送は，

ストリーム <-- 式

によって行なう．ここで，サーバ側で要求の受信から結果の返送が行なわれるまでは，他のプロセスからのサーバへの要求の送信は中断され，結果の返送後，中断しているプロセスの実行が再開される．

3.2.2 記述例

ここでは，SSDL によるプロセス記述の例を示す．まず，ストリームによるパイプライン型の並列処理の例として，エラトステネスのふるいによる素数生成を行なう並列プログラムの記述を示し，アトミック・ストリームを用いたクライアント・サーバ型のプロセス記述の例として，渡された数値の階乗を計算するサーバプロセスの記述の例を示す．また，select ステートメントによる非決定的なストリーム入力の例としてストリームのマージの記述を示す．

(1) 素数生成

ここでは，SSDL による動的なプロセス生成と，ストリームによるパイプライン型の並列処理の例として，エラトステネスのふるいにより素数生成を行なうプログラムの例を示す．このプログラムはまた，動的なプロセスの生成とストリームを介したストリームの受渡しによる動的なプロセスの結合ネットワークの構築の例ともなっている．以下の定義で，4 行目で定義されるプロセス gen は，引数 from と to で指定された範囲の整数列を，7 ～ 10 行の for 文によって出力ストリーム int_s に出力し，11 行目のストリームへの出力式によってストリームを閉じ，実行を終了する．15 行目で定義されるプロセス shift は，入力ストリーム int_s の最初の要素である素数 v に対して，int_s の残りの要素をふるいにかけるプロセス filter と，filter の出力を素数列へと変換とするプロセス shift の生成を 21 ～ 24 行の process ステートメントにより行ない，素数 v をストリーム pri_s に出力した後，26 ～ 29 行の while 文により先に生成したプロセス shift からストリーム res_s を介して送られてくる素数列をストリーム pri_s へ中継する．プロセス shift は入力ストリーム int_s が実行開始時に閉じられているか，ストリーム res_s が閉じられると，出力ストリーム pri_s を閉じ実行を終了する．34 行で定義されるプロセス filter は，37 ～ 41 行の while 文により入力ストリーム int_s から受けとっ

た整数列に対して，引数 n で指定された素数によってふるいを行ない，結果を出力ストリーム fil_s に出力する．プロセス filter は入力ストリーム int_s が閉じられると，出力ストリーム fil_s を閉じ，実行を終了する．プロセス main は，プログラムが実行を開始すると共に自動的に生成され，49 ～ 52 行の process 文により 2 から N までの整数列を出力するプロセス gen と，これに対する素数列を生成するプロセス shift を生成することによってプロセスの初期化を行ない，その後生成したプロセス shift から送られてくる素数列の表示を行なう．

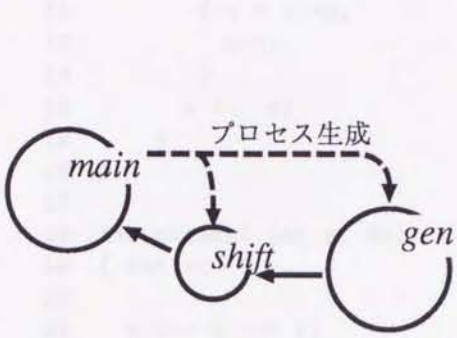
```
 1  // -*- c -*-
 2  Const N = 3000;
 3
 4  int_s <- gen( int from, int to)
 5  { int i;
 6
 7    for( i = from; i <= to; i++)
 8      {
 9        int_s <- i;
10      }
11    int_s <- [];
12  }
13
14
15  pri_s <- shift( int_s)
16  { int v, n;
17    Stream fil_s, res_s;
18
19    if( int_s -> v)
20      {
21        process {
22          fil_s <- filter( v, int_s);
23          res_s <- shift( fil_s);
24        }
25        pri_s <- v;
26        while( res_s -> n)
27          {
28            pri_s <- n;
29          }
30      }
31    pri_s <- [];
32  }
33
34  fil_s <- filter( int n, int_s)
35  { int v;
36
37    while( int_s -> v)
38      {
39        if( v % n != 0)
40          fil_s <- v;
41      }
42    fil_s <- [];
43  }
44
```

```
45  main()
46  { int n, i;
47    Stream int_s, pri_s;
48
49    process {
50      int_s <- gen( 2, N);
51      pri_s <- shift( int_s);
52    }
53    printf( "[");
54    for( i = 0; pri_s -> n; i++)
55      { printf( " %d,", n);
56      }
57    printf( "]/%d primes.\n", i);
58  }
59
```

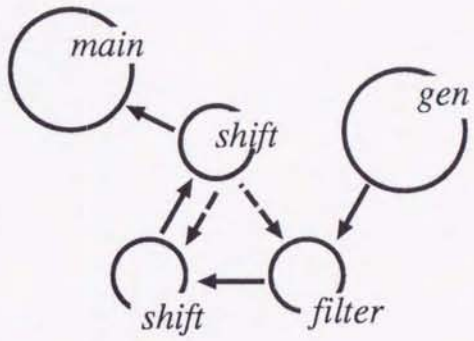
このプログラムにおける，プロセスの動的な生成の様子を図 3.4 に示す．この図では，プロセスを白抜きの丸で，これらのプロセス間を結合するストリームをデータの転送方向に対応した実線の矢印で表している．また，プロセスの動的な生成を破線で示している．ここで，初期状態 (a) はプロセス main が実行を開始し，2 から定数 N で定義される値までの整数列を生成するプロセス gen と，フィルタプロセスの生成の制御を行なうプロセス shift を生成した時点の状態を表している．状態 (a) で生成されたプロセス shift が，入力される数値列の先頭要素に対応してフィルタを行なうプロセス filter と，入力ストリームの残りの要素に対する shift プロセスを生成することによって，状態 (a) は状態 (b) へ移行する．さらに処理を継続することによって，最終的には (c) に示すようなプロセスのネットワークが構成される．状態 (c) では N までに存在する素数の数を m 個とすると，各素数に対応して m 個のプロセス filter が，gen から順にパイプライン的に結合されてフィルタ処理を行ない，さらに m+1 個の shift プロセスが同様にパイプライン的に結合されて素数列の中継処理を行なう．

(2) 階乗計算

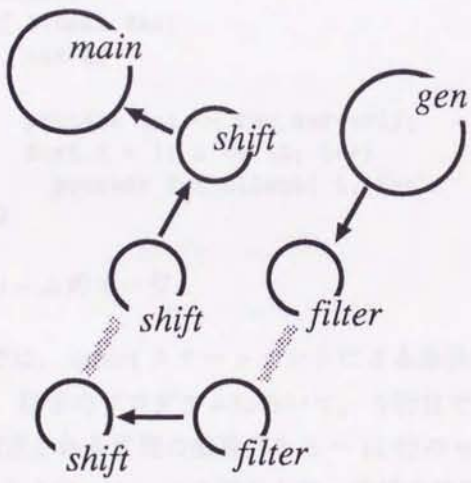
アトミック・ストリームによるクライアント・サーバ型のプロセス記述の例として，階乗を計算するサーバ・プロセスとクライアント・プロセスの記述を示す．3 行目で定義されるプロセス fac_server は，アトミックス・トリーム s から階乗計算の要求を表す数値 n を 7 行目のアトミック・ストリーム入力式で受けとると，その階乗を計算し，結果をアトミック・ストリーム s を通して 14 行目のアトミック・ストリーム出力式でクライアント・プロセスへ返却する処理を，6 ～ 15 行の while 文によって繰り返す．プロセス fac_client は，引数で与えられた数 n に対する階乗の計算要求を，アトミック・ストリーム s を介してサーバ・プロセスに送るとともに結果を変数 v に受けとり，結果を表示した後実行を終了する．ここで，複数の fac_client から fac_server への通信は，アトミック・ストリームへの送受信によって排他制御が行なわれる．25 行目で定義されるプロセス main は，29 行目の process 文でサーバ・プロセス fac_server の生成を行ない，30 ～ 31 行目の for 文によって 1 から 10 までの各整数値に対して素数の表示を行なうクライアント・プロセスの生成を行ない，実行を終了する．



(a) 初期状態



(b) フィルタプロセス生成後



(c) 最終状態

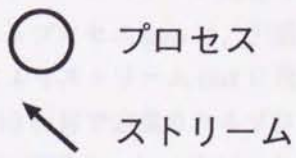


図 3.4 素数生成プログラムによるプロセス生成の様子

```
 1  // -*- c -*-
 2
 3  @s <- fac_server()
 4  { int m, n, v;
 5
 6    while( 1)
 7      { s --> n;
 8        v = 1;
 9        m = n;
10        while( m > 1)
11         { v = v *m;
12           m--;
13         }
14        s <-- v;
15      }
16  }
17
18  fac_client( int n, @s)
19  { int v;
20
21    s <-- n --> v;
22    printf( "fac( %d) =  %2d\n", n, v);
23  }
24
25  main()
26  { Atomic fac;
27    int i;
28
29    process fac <- fac_server();
30    for( i = 1; i <= 10; i++)
31      process fac_client( i, fac);
32  }
```

(3) ストリームのマージ

ここでは，selectステートメントによる非決定的なストリームのマージ処理の記述例を示す．以下のプログラムにおいて，5行目で定義されるプロセスgenは，引数fromとtoで指定される区間の整数列を9～12行のwhile文によりストリームoutに出力し，13行目でストリームoutを閉じた後，実行を終了する．16行目で定義されるプロセスmergeは，入力ストリームin1およびin2からの入力を23行目のselect文により非決定的に選択し，選択されたストリームから受けとった値をストリームoutに出力する処理を21～37行のwhile文により繰り返す．ここでは，各ストリームが閉じられた場合に対応するため，ストリームの開閉状態をストリームin1およびin2それぞれに対応した変数open1およびopen2に記憶しておき，select文によるストリームの選択を制御するとともに，ストリームが双方とも閉じられた場合に，出力ストリームoutを閉じる操作を行なった後，実行を終了する．41行で定義されるプロセスmainは，0から20および100から120の整数列を生成する2つのプロセスgenと，これらのプロセスからの出力を非同期的にマージするプロセスmergeの生成を45～49行のprocess文により行ない，50～51行のwhile文によりマージされた結果を表示する．

```
 1  // -*- c -*-
 2
 3  Const TRUE = 1, FALSE = 0;
 4
 5  out <- gen( int from, int to)
 6  { int i;
 7
 8    i = from;
 9    while( i <= to)
10      {
11        out <- i++;
12      }
13    out <- [];
14  }
15
16  out <- merge( in1, in2)
17  { int v;
18    int open1, open2;
19
20    open1 = open2 = TRUE;
21    while( 1)
22      {
23        select {
24          ( open1 ) && in1 -> v
25            {
26              out <- v;
27            }
28          : open1 = FALSE;
29          ( open2 ) && in2 -> v
30            {
31              out <- v;
32            }
33          : open2 = FALSE;
34        }
35        if( !open1 && !open2)
36          break;
37      }
38    out <- [];
39  }
40
41  main()
42  { Stream s1, s2, m;
43    int v;
44
45    process {
46      s1 <- gen( 0, 20);
47      s2 <- gen( 100, 120);
48      m  <- merge( s1, s2);
49    }
50    while( m -> v)
51      printf( "%d\n", v);
52  }
```

3.3 変換系

SSGL および SSDL の記述は変換系によって，YACC による定義，C 言語による定義などへ変換される．本節では，これら変換系の実現について簡単に説明する．

3.3.1 SSGL 変換系

SSGL による意味処理の記述は，3.1 節で説明したように目的とする言語の構文規則と共に与え，SSG と呼ばれる生成系によって，LEX[Lesk 75] 定義，YACC 定義などが生成される．これらは，さらに構文解析部へのデータ，意味解析プロセス生成のためのデータと，各意味解析プロセスを生成するために必要なプログラムとデータへと変換される．

ここで，SSGL 記述において各生成規則に付随して記述される意味規則は，生成される YACC 定義においてストリームとプロセスの生成，ストリームの結合を行なう動作ルーチンへと変換される．なお，3.3.1 項で触れたように，SSGL 記述では意味処理を行なう仮想的なプロセスを動的に統合する機構を用意しているが，以下では，独立した実体として生成される意味解析プロセスの変換後の扱いのみに関して説明を行なう．逐次化対象のプロセスの扱いに関しては，5 章および 6 章で説明する．

意味規則の変換は，各意味規則において，'=' の左辺に現れるストリームを生成し構文解析スタックと同期して動作する意味解析スタック上に保存するための動作ルーチンを，そのストリームが定義された非終端記号の直前に配置し，右辺に現れるプロセスに対して，そのプロセスが出力するストリームが定義された非終端記号の直前，あるいは，プロセスが入力するストリームが定義された文法記号の直後のうち，生成規則上で最後に現れる位置にプロセスを生成する動作ルーチンを配置することにより行なう．プロセスの生成に対応する動作ルーチンは，構文解析器の状態の衝突を減少させるため，生成時のオプションによって生成規則の最後尾に配置することも可能である．プロセス間を結合するために必要なストリームは意味解析スタック上に保存されているため，プロセス生成時にこのスタックを介して生成規則間でストリームの受渡しが行なわれ，プロセスの結合が行なわれる．

ここで，

```
ストリーム = ストリーム;
```

で表されるストリームからストリームへのコピー規則に関しては，右辺に現れるストリームの定義された文法記号が，左辺のストリームの定義された文法記号より左に現れる場合と，右辺に現れるストリームがその生成規則で定義される文法記号に属したものである場合に関しては，ストリームの生成とストリーム間のコピーを行なうプロセスの生成は行なわず，ストリームを参照するポインタのコピーが行なわれる．

例えば，

```
1  While_Statement
2          : WHILE Cond DO Statement
3                  { [ While_Statement.result, Cond.exec, Statement.exec]
```

```
                    = while_stmt( While_Statement.exec,
                                  Cond.result, Statement.result);
                Cond.symtab = While_Statement.symtab;
                Statement.symtab = While_Statement.symtab;
              }
    ;
```

のようなSSGL記述は，ほぼ次のような処理に対応する動作ルーチンに変換される．

```
While_Statement
    : WHILE     { Cond.exec   = create_stream();
                  Cond.symtab = While_Statement.symtab;
                }
      Cond DO   { Statement.exec = create_stream();
                  Statement.symtab = While_Statement.symtab;
                }
      Statement { While_Statement.result = create_stream();
                  Create_New_Process {
                    [ While_Statement.result,
                      Cond.exec, Statement.exec]
                          <- while_stmt( While_Statement.exec,
                                         Cond.result,
                                         Statement.result);
                  }
                }
    ;
```

ここで，create_stream() は新たなストリームを生成する関数とし，Create_New_Processは，

[...,出力ストリーム,...] <- プロセス名 (...,入力ストリーム,...)

という形式で新たなプロセスの生成を行ない，ストリームによる結合を行なうものとする．

SSGL記述からYACC記述の生成時には，生成されたYACC記述から構文解析器を生成する際の解析状態の衝突を減少させるため，変換後の動作ルーチンが属性スタック上の変数の参照時のスタック変移を含めて同一のものに関して，これらを1つの動作ルーチンに統合する処理を行なうことができる．

3.3.2 SSDL変換系

本節では，SSDLの変換系の実現について簡単に説明する．SSDLは記述は，SSDCと呼ばれる変換系によってC言語のプログラムに変換される．ここでSSDL記述のうち，プロセスの定義はプロセスの生成手続きおよびプロセス本体に対応する手続きへと変換され，出力ストリームおよび入力パラメータは，これらの手続きの引数として扱われる．SSDLのステートメントおよび式の構文ははほぼC言語の構文と同等であり，これらは等価なC言語の表現に変換される．プロセスに局所的な変数に関しては，整数型および文字型の変数については，対応するC言語の変数宣言にそのまま変換される．ストリーム型，アトミック・ストリーム型および外部型に関しては対応するC言

語の型のポインタ型へと変換される．ただし，ストリーム型およびアトミック・ストリーム型に関しては，ポインタ形式でない宣言に関しては実体の割り当てと変数への代入が自動的に行なわれる．例えば，

```
Stream s1, *s2;
```

という宣言では，s1 には実体が割り当てられることになるが，s2 には実体が割り当てられない．プロセスの生成と，ストリームによる通信を行なうステートメントおよび式は，変換後のプログラム中では 5 章で示す仮想マシン・プリミティブの呼びだしに変換される．これらのプリミティブは，プロセスの動的な生成，ストリームへの送信と受信，ストリーム中のデータの確認，アトミックなデータの送受信を行なうプリミティブ等から構成される．

第 4 章

コンパイラの意味処理の記述

言語処理系における典型的な意味処理は，大きくわけてコンパイラなどにみられる静的な意味処理と，インタプリタなどの実行時の意味に対応する動的な意味処理に分類することが可能である．ここでは，コンパイラなどにおける意味処理のストリームに基づいた記述の具体的な例として，記号表管理の実現と，コンパイラにおける中間コード生成を対象として，SSGL および SSDL による記述を示し．同時に，その効果的な実現のための技法を示す．

4.1 中間コード生成

本節では while 文の中間コード生成の SSGL 記述を示す．以下の定義で，code は中間コードの受渡しを行なうためのストリーム，symtab は記号表管理を行なうプロセスと，記号表参照を記号表参照を行なうプロセスとの間で双方向の通信を行なうためのアトミック・ストリームである．この記号表の参照方式に関しては次節で説明する．

プロセス while_stmt は，条件式およびステートメントの中間コードに対応するストリーム Cond.code および Statement.code を引数としてとり，while 文に対応する中間コードをストリーム While_Statement.code に出力する．

```
1  While_Statement
2          : WHILE Cond DO Statement
3                  { While_Statement.code
4                          = while_stmt( Cond.code, Statement.code);
5                    Cond.symtab = While_Statement.symtab;
6                    Statement.code = While_Statement.symtab;
7                  }
8          ;
```

(1) コピー型の中間コード生成プロセスの定義

以下に，プロセス while_stmt の定義を示す．ここでで new_label は一意な名称のラベルを生成する関数，new_code は第一引数で指定された種類の中間コードを生成する関数である．この定義では，while 文の制御に相当する中間コードの空隙に，条件式およびステートメントに対応する中間コードをコピーすることにより挿入し，while 文の

中間コードの生成を行なう．具体的には，プロセス while は，まず 5 ～ 6 行目で while 文のボディーと条件部それぞれに対応する一意なラベルを生成し，変数 body_label および cond_label に格納する．続いて，7 ～ 8 行で，条件式への無条件分岐を行なうコードと，ボディー部のラベルに対応する中間コードを生成し，ストリーム self_code へ出力する．9 ～ 10 行の while 文では，while 文のボディーに対応する中間コードを，ストリーム stmt_code から self_code にコピーする．続いて，11 行目では条件部に対するラベルを self_code に出力し，12 ～ 13 行目の while 文で条件式に対応する中間コードをストリーム cond_code から self_code へコピーする．次に，14 行目では条件部の結果によってボディーへの条件分岐を行なう中間コードを，self_code へ出力する．最後に，15 行目でストリーム self_code を閉じた後，実行を終了する．

```
 1  self_code <- while_stmt( cond_code, stmt_code)
 2  { LABEL body_label, cond_label;
 3    CODE code;
 4
 5    body_label = new_label();
 6    cond_label = new_label();
 7    self_code <- new_code( GOTO,  cond_label);
 8    self_code <- new_code( LABEL, body_label);
 9    while( stmt_code -> code)
10      self_code <- code;
11    self_code <- new_code( LABEL, cond_label);
12    while( cond_code -> code)
13      self_code <- code;
14    self_code <- new_code( IF_TRUE, body_label);
15    self_code <- [];
16  }
```

ここでの while 文に対する中間コード生成の様子を図 4.1 に示す．図 4.1 では破線によって while 文に対応する解析木の構造を，白抜きの丸によって while 文の中間コードを行なうプロセスを表している．中間コードに対応するストリームは中間コードの転送方向を示す単方向の矢印で示し，while 文の条件部およびボディー部に割り当てられるプロセスと記号表の管理を行なうプロセス間の通信に使用されるストリームを双方向の矢印で示している．さらに，while 文に対応する中間コードを '[', ']' で括って示している．この図に示すように while 文の中間コード生成を司るプロセスは，while 文の構造に従って，条件部およびボディー部の中間コード生成を行なうプロセスとストリームによって結合され，上位ノードに対して生成した中間コードを出力する．while 文に対応するプロセスは，条件部およびボディー部の中間コードをストリームから受けとり，条件部への分岐，ボディー部のラベル，ボディー部に対応するステートメントの中間コード列，条件部のラベル，条件部の中間コード列，ボディー部への条件分岐をこの順に出力する．

(2) ストリーム転送型中間コード生成プロセスの定義

次のようにストリーム自身をストリームを介して受け渡すことにより，中間コード生成処理の定義を簡略化することも可能である．以下の定義では，上の定義でストリー

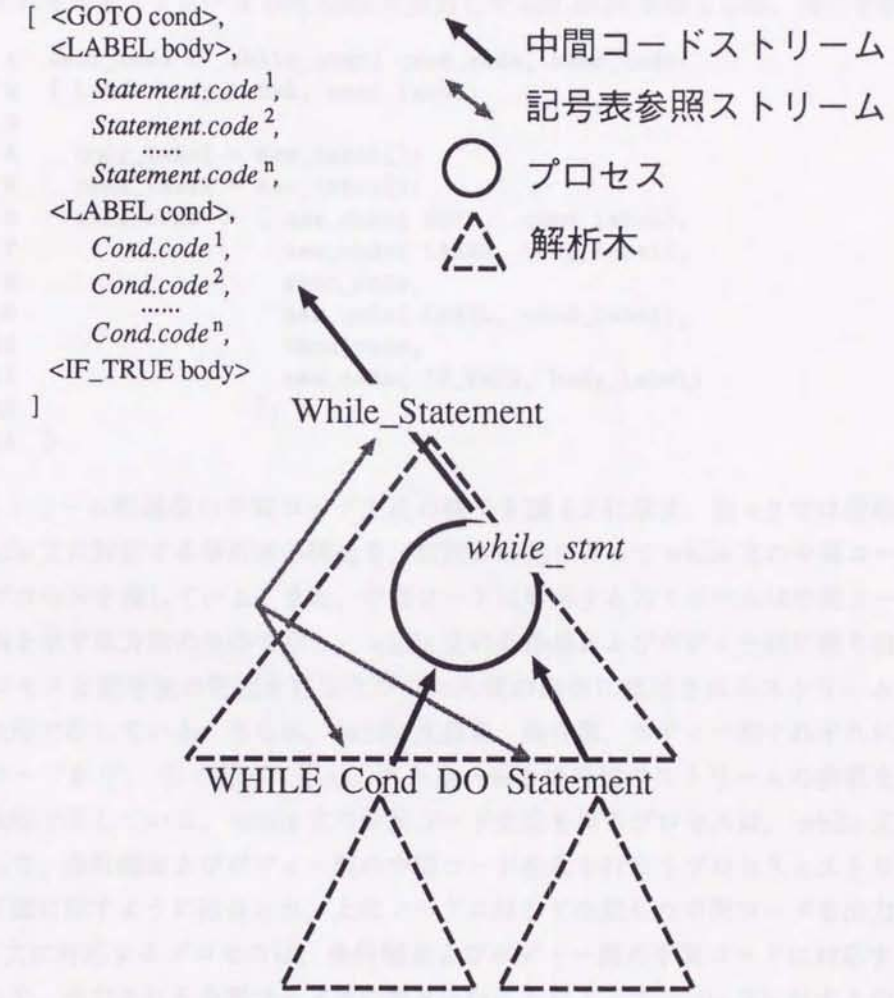


図 4.1 コピー型のコード生成

ムからストリームへのコピーを行なっていた条件式およびステートメントに対応する中間コードに対応するストリームを，直接プロセス while_stmt の出力ストリーム self_code に出力する．具体的にはプロセス while ステートメントは，上の定義と同様に，4 ～ 5 行で while 文のボディーと条件部それぞれに対応する一意なラベルを生成する．続いて，6 ～ 12 行のストリームへの出力式によって，条件部への無条件分岐を行なうコード，ボディー部のラベルのコード，ボディに対応したストリーム stmt_code，条件部のラベルのコード，条件部に対応したストリーム cond_code，ボディーへの無条件分岐のコード，それぞれをストリーム self_code に出力して self_code を閉じた後，実行を終了する．

```
 1  self_code <- while_stmt( cond_code, stmt_code)
 2  { LABEL body_label, cond_label;
 3
 4    body_label = new_label();
 5    cond_label = new_label();
 6    self_code <- [ new_code( GOTO,  cond_label),
 7                   new_code( LABEL, body_label),
 8                   stmt_code,
 9                   new_code( LABEL, cond_label),
10                   cond_code,
11                   new_code( IF_TRUE, body_label)
12                 ];
13  }
```

ストリーム転送型の中間コード生成の様子を図 4.2 に示す．図 4.2 では破線によって while 文に対応する解析木の構造を，白抜きの丸によって while 文の中間コードを行なうプロセスを表している．また，中間コードに対応するストリームは中間コードの転送方向を示す単方向の矢印で示し，while 文の条件部およびボディー部に割り当てられるプロセスと記号表の管理を行なうプロセス間の通信に使用されるストリームを双方向の矢印で示している．さらに，while 文自身，条件部，ボディー部それぞれに対する中間コードを '[', ']' で括って示し，ストリーム中での他のストリームの参照を斜線による矢印で示している．while 文の中間コード生成を司るプロセスは，while 文の構造に従って，条件部およびボディー部の中間コード生成を行なうプロセスとストリームによって図に示すように結合され，上位ノードに対して生成した中間コードを出力する．while 文に対応するプロセスは，条件部およびボディー部の中間コードに対応するストリームを，出力される中間コード列に挟み込むことによって while 文に対する中間コードを構成する．ここで，構成される中間コード列は，条件部への分岐，ボディー部のラベル，ボディー部に対応するストリーム，条件部のラベル，条件部に対応するストリーム，ボディー部への条件分岐からなる．

(3) 中間コードコピーを行なうプロセスの削除

生成規則の部分解析木の中間コードに対応するストリームを直接ストリームを介して受け渡すことは，記述の簡略化という面だけでなく，中間コードのコピーを行なうプロセスを削除し，同時に存在するプロセスの数を減少することが可能であることを意味している．

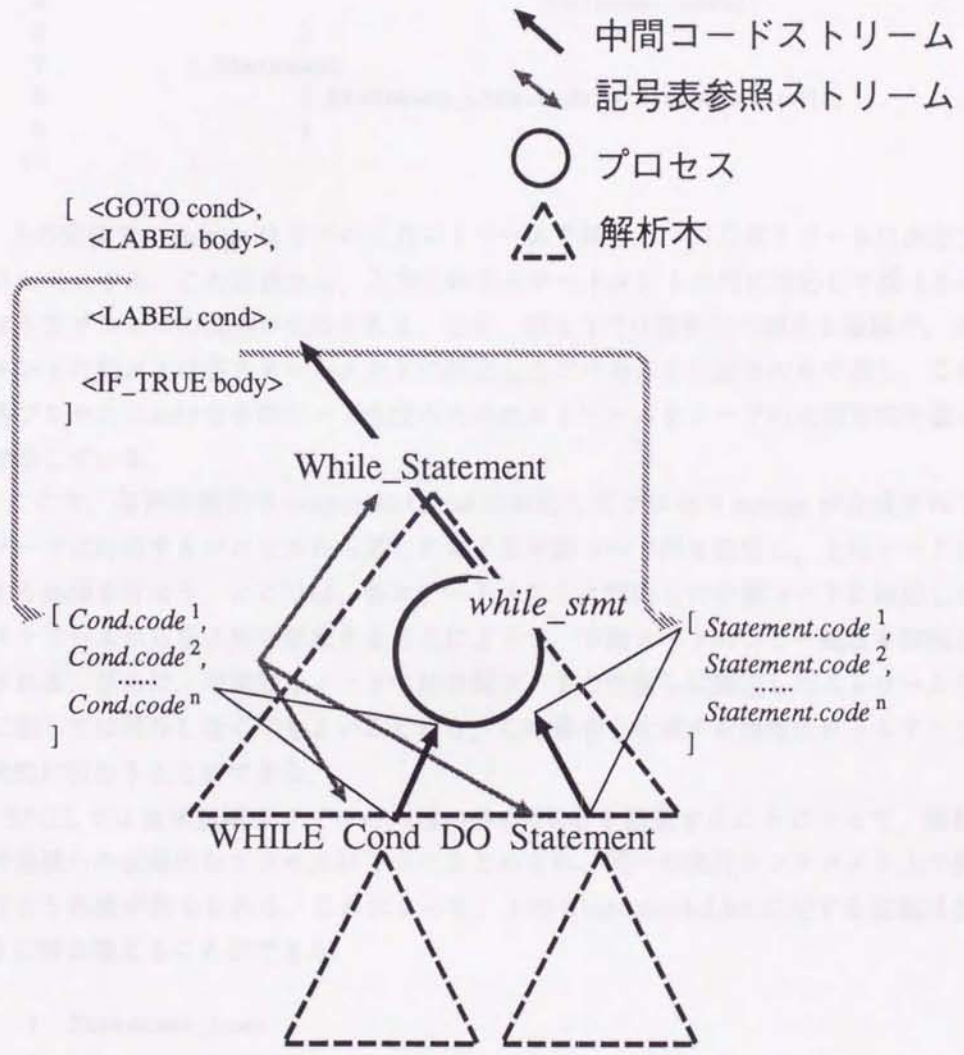


図 4.2 ストリーム転送型のコード生成

例えば，非終端記号 While_Statement の上位ノードとなる Statement_List の定義を考える．Statement_List に対応する SSGL の中間コード生成に関連した定義は次のように与えられる．

```
1  Statement_List
2          : Statement_List Statement
3                  { Statement_List[0].code
4                          = merge( Statement_List[1].code,
5                                   Statement.code);
6                  }
7          | Statement
8                  { Statement_List.code = Statement.code;
9                  }
10         ;
```

上の定義で，merge は 2 つの入力ストリームを結合して出力ストリームに出力するプロセスとする．この定義から，入力されるステートメントの列に対応して図 4.3 に示すようなプロセスの構造が生成される．なお，図 4.3 では解析木の構造を破線で，ステートメントの列および各ステートメントに対応したプロセスを白抜きの丸で表し，これらの各プロセスにおける中間コード生成のためのストリームをコードの送信方向を表す矢印で示している．

ここで，各非終端記号 Statement_List に対応してプロセス merge が生成され下位のノードに対応するプロセスから送られてくる中間コード列を合成し，上位ノードに送信する処理を行なう．ここでは，各ステートメントに対応した中間コードに対応した出力ストリームからなる列を構成することによって，中間コードのコピー処理を抑制が抑制される．さらに，中間的なノードでは中間コードの受渡しに関連したストリーム中の値に関しては関与しなくてもよいことから，この構造を生成する処理はボトムアップに逐次的に行なうことができる．

SSGL では意味規則中でプロセス名の前に SEQ を前置することによって，解析木上で連続した仮想的なプロセスが 1 つにまとめられ，同一の実行コンテクスト上で実行を行なう処理が行なわれる．これによって，上の Statement_List に対する定義は次のように書き換えることができる．

```
1  Statement_List
2          : Statement_List Statement
3                  { Statement_List[0].code
4                          = SEQ smerge( Statement_List[1].code,
5                                        Statement.code);
6                  }
7          | Statement
8                  { Statement_List.code = Statement.code;
9                  }
10         ;
```

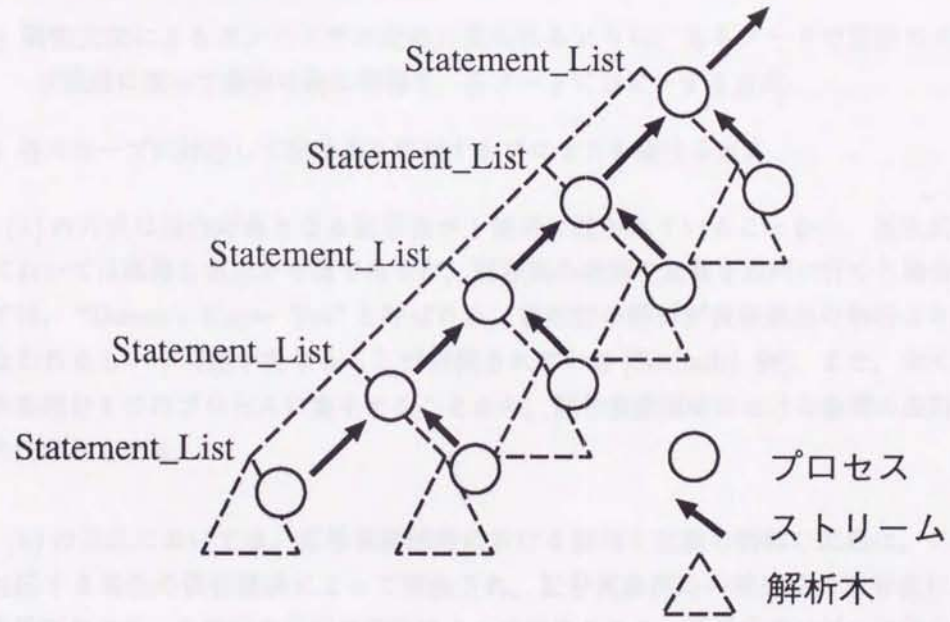


図 4.3 ステートメント・リストに対するプロセスの構造

ここで，smergeは引数の2つのストリーム自身を要素とする構造を作成する関数であり，これを解析木で指定される構造に従ってボトムアップに実行することによって，中間レベルでのコードの中継を行なうプロセスを1つのプロセスに統合することができる．

4.2 記号表参照

ここでは，コンパイラにおける記号表参照の記述を示す．
記号表参照の記述には，以下のように幾つかの方式が考えられる．

(a) 手書きのコンパイラに見られるような，1つのプロセスで全ての情報を管理する方式

(b) 属性文法によるコンパイラの記述に見られるように，あるノードで言語のスコープ規則に従って参照可能な情報を，各ノードにコピーする方式

(c) 各スコープに対応して記号表を管理するプロセスを設ける方式

(a)の方式は操作対象となる記号表が1箇所に置かれていることから，逐次的な実現においては高速な実現が可能であるが，記号表の参照や定義を並列に行なう場合においては，“Doesn't Know Yet”と呼ばれる，参照部の解析が被参照部の解析より先に行なわれるという問題が生じることが指摘されている[Seshadri 88]．また，全ての記号表参照が1つのプロセスに集中することから，記号表参照時における参照の並列性抽出の妨げともなる．

(b)の方式においては，記号表参照時における参照と定義の制御の問題は，これらに対応する属性の依存関係によって解決され，記号表参照時の並列性は記号表に対応する属性のコピーの参照の並列な実行によって抽出される．記号表のコピーに伴う問題はポインタによる実体の共有で解決可能であるが，各スコープを構成する記号表の構築は，スコープの外側から内側に向かって順に行なわれる必要がある．このため，非局所的な名前の参照を行なわないようなブロックの処理に関しても，スコープルールに従った順序での記号表構造の構成と情報のコピーが必要となる．また，定義エラーの発生時には記号表内にエラーに関する情報を加える技法が有効であるが，記号表のコピーによる方式ではこのような技法を導入することが難しい．

(c)の方式は基本的には(b)の方式と同様であるが，局所的な参照に関しては，外側のスコープの記号表の構築以前に行なうことが可能である．また，論理的な宣言に対応した記号表の情報が1箇所で管理されることから，エラー処理時の記号表内の情報の書き換えを行なうことも可能である．ただし，記号表を単一のプロセスによって管理することによって，同一のスコープの記号表に対する参照の並列度が制限される．

ここでは (c) のように各宣言スコープに対応して記号表管理を行なうプロセスを設ける方式での実現と，その効果的な実現のための方式を示す．以下は，この方式による変数宣言部の SSGL 記述の例である．ここで，symtab は記号表参照を行なうためのアトミック・ストリーム，info は記号表に保存される情報，list は記号表に保存される情報のリスト，size は局所ブロックに対応したフレームの大きさ，name は識別子に対応する文字列に対応している．なお，ここでは，size, list, info を構築するプロセスは SEQ で指定され逐次的に評価が行なわれ，列的な構造を持たないデータが生成される．

```
 1  Var_Dcl_Part
 2          :               { Var_Dcl_Part.symtab = Var_Dcl_Part.symtab;
 3                            Var_Dcl_Part.size   = 0;
 4                          }
 5          | VAR Var_Dcl_List ';'
 6                          { Var_Dcl_Part.symtab
 7                                  = var_symtab( Var_Dcl_Part.symtab,
 8                                                Var_Dcl_List.list);
 9                            Var_Dcl_Part.size
10                                  = SEQ count_list( Var_Dcl_List.list);
11                          }
12          ;
13  Var_Dcl_List
14          : Var           { Var_Dcl_List.list
15                                  = SEQ make_list( Var.info, NULL);
16                          }
17          | Var ',' Var_Dcl_List
18                          { Var_Dcl_List[ 0].list
19                                  = SEQ make_list( Var.info,
20                                                   Var_Dcl_List[ 1].list);
21                          }
22          ;
23  Var
24          : ID            { Var.info = SEQ make_var_info( ID.name);
25                          }
26          ;
```

変数に対する記号表の管理プロセス var_symtab の SSDL 定義を以下に示す．まず，プロセス var_symtab は引数で与えられる宣言のされた変数名のリスト var_list に対して，3行目で手続き alloc_var_symtab を呼び出して局所的な記号表を構築する．ここで，alloc_var_symtab は同時に変数のフレーム上の割り当てなどの処理も行なう．続いて，var_symtab は6行目のアトミックストリーム入力式によってアトミック・ストリーム symtab からの記号表参照要求を待ち，7行目で手続き sym_access を呼び出すことによって局所的な記号表の参照を行なう．局所的な記号表に定義が存在しない場合は，10行において，記号表参照用のアトミック・ストリーム outer_symtab を介してより外側の記号表管理プロセスに対して参照を行なう．

SSGL では，ストリームを介して受け渡すことが可能なものは，基本型かポインタである．このため，以下の実現では，記号表によって管理される情報に関しては共有メモリ上に確保し，記号表検索のための表構造自身に関しては局所メモリ上に確保する．

```
1  @symtab <- var_symtab( @outer_symtab, List var_list)
2  { INFO info, ret;
3    SYMTAB local_symtab = alloc_var_symtab( var_list);
4
5    while( 1)
6      { symtab --> info;
7        if( ret = sym_access( local_symtab, info))
8          symtab <-- ret;
9        else
10         { outer_symtab <-- info --> ret;
11           symtab <-- ret;
12         }
13     }
14 }
```

解析木上での，変数宣言部に対する記号表管理を行なうプロセスは図 4.4 のように与えられる．ここで，破線は解析木の部分的な構造を示す．白抜きの丸は変数宣言に対する記号表管理を行なうプロセスを示し，プロセス内の局所データとして管理される記号表を矩形で示す．また，変数宣言部で集められる名前に関する情報の参照を単方向の矢印で，記号表の参照と結果の返送を行なうストリームを双方向の矢印で示している．図に示すように，変数宣言に対する記号表の管理を行なうプロセスは，宣言された名前の情報を受けとるとともに，より外側の記号表を管理するプロセスおよび当該記号表の参照を行なうプロセス間と双方向のストリームによって結合される．記号表管理プロセスは，この双方向のストリームから参照要求を受けとり，結果の返送を行なう．要求に対する宣言が存在しない場合は，より外側の記号表を管理するプロセスに参照要求を送信する．

ブロック構造を持つ言語では，非局所的な定義の参照が頻繁おこなわれる，また，上の定義では変数の定義や記号定数の定義などに個別の記号表管理プロセスを設けており，非局所的な記号の参照を行なう場合にはプロセス var_symtab による記号表の参照要求と結果の中継が行なわれることになる．そこで，次のように上の定義を書き換えることにより記号表参照のキャッシングを行なうように変更することができる．ここで，register は局所的な記号表 local_symtab に対して，より外側の記号表管理プロセスの参照結果 ret を新たに登録する手続きであり，より外側の記号表に対する参照を 10 行目で行なった後，その結果を register を呼びだすことによってキャッシングを行なう．このように，記号表の情報のキャッシングを局所的な記号表参照プロセス内で行なうことによって，非局所的な記号表参照に伴うプロセス間通信回数を減少させることができる．

```
1  @symtab <- var_symtab( @outer_symtab, List var_list)
2  { INFO info, ret;
3    SYMTAB local_symtab = alloc_var_symtab( var_list);
4
5    while( 1)
6      { symtab --> info;
7        if( ret = sym_access( local_symtab, info))
8          symtab <-- ret;
```

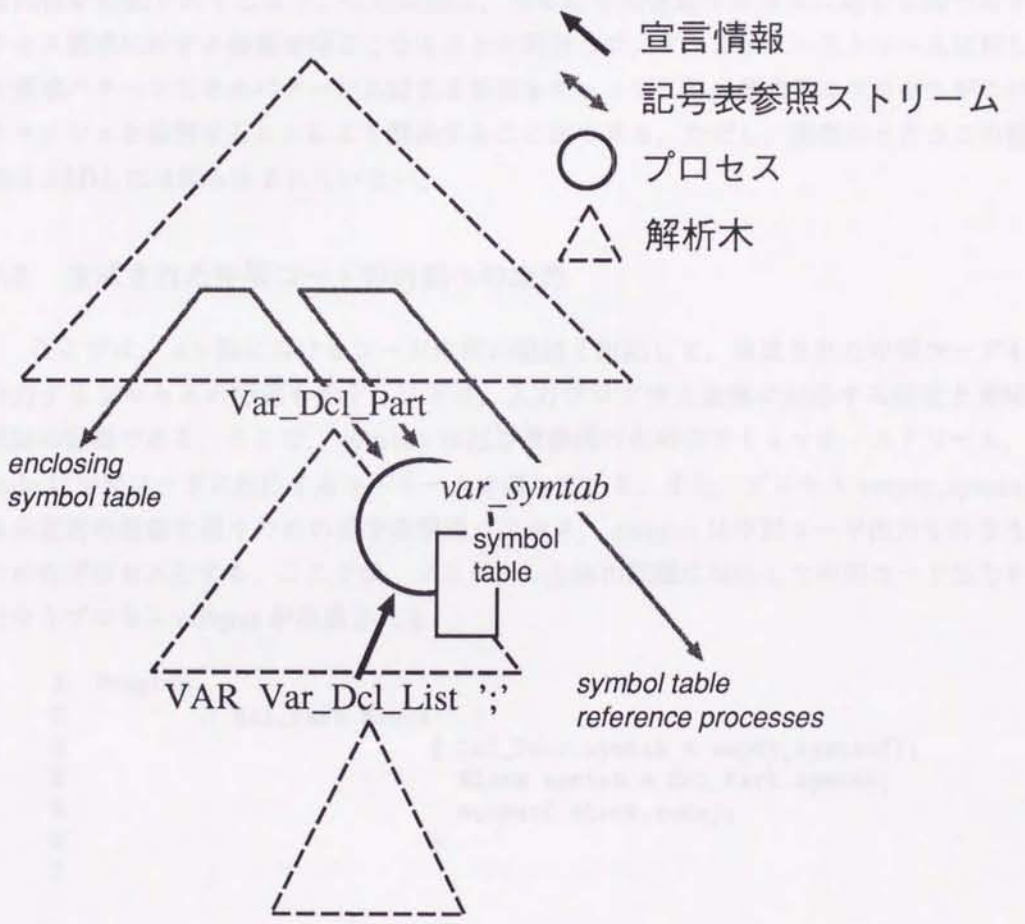

宣言情報
記号表参照ストリーム
プロセス
解析木
Var_Dcl_Part
enclosing
symbol table
var_symtab
symbol
table
VAR Var_Dcl_List ';'
symbol table
reference processes

図 4.4 記号表管理

```
 9        else
10          { outer_symtab <-- info --> ret;
11            register( local_symtab, ret);
12            symtab <-- ret;
13          }
14      }
15  }
```

ここで示した方式では，同一の記号表アクセスに対する複数の参照要求は，アトミック・ストリームに対する送信によって排他制御が行なわれるため，記号表参照に伴う並列性が制約されてしまう．この問題は，ある記号表管理プロセスに対する同一のアクセス要求に対する結果が同じとなることを利用して，アトミック・ストリームに対して要求パターンとそのパターンに対する結果をキャッシュし，要求側のプロセスがこのキャッシュを参照することにより解決することができる．ただし，現在のところこの機能はSSDLには組み込まれていない．

4.3 生成された中間コードの外部への出力

ここでは，4.1節におけるコード生成の記述と対応して，生成された中間コードを出力するプロセスの記述を示す．以下は，入力プログラム全体に対応する構文と意味規則の定義である．ここで，symtabは記号表参照のためのアトミック・ストリーム，codeは中間コードに対応するストリームを表している．また，プロセスempty_symtabは未宣言の名前を扱うための記号表管理プロセス，outputは中間コード出力を行なうためのプロセスとする．ここでは，プログラム全体の認識に対応して中間コード出力を行なうプロセスoutputが生成される．

```
1  Program
2          : Dcl_Part Block '.'
3                          { Dcl_Part.symtab = empty_symtab();
4                            Block.symtab = Dcl_Part.symtab;
5                            output( Block.code);
6                          }
7          ;
```

次は，中間コードを各ノードで上位ノードにコピーするタイプの実現に対する中間コード出力プロセスの実現である．この記述では，4～5行のwhile文によりストリームcode_streamから中間コードを受けとり，C言語の関数print_codeを呼び出してコードの出力を行なう．

```
1  output( code_stream)
2  { CODE code;
3
4    while( code_stream -> code)
5      { print_code( code);
6      }
7  }
```

以下は，(1) で示したストリームを介してストリームを受け渡すタイプの実現でのプロセス output の記述である．ここでは，再帰的に定義されたストリームからのデータを受信するため，入力が中断されているストリームを保存するスタック stream_stack を使用する．プロセス output は 8 行目のストリーム入力式によってストリーム code_stream から中間コードまたは中間コードのストリームを入力する．入力データがストリームであった場合，そのストリームのアドレスがストリームへのポインタ new_stream へ代入されるので，12 ～ 13 行目で code_stream をスタック stream_stack にプッシュすると共に，code_stream が new_stream の示すストリームを指すようにする．入力が中間コードの場合，16 行目で print_code を呼び出してこれを出力する．このストリーム中にデータが存在せず，閉じられている場合には，22 行目でスタック stream_stack からストリームを取り出し，code_stream へ代入する．スタックが空の場合にはコードの出力処理を終了する．

```
 1  output( code_stream)
 2  { CODE code;
 3    Stream *new_stream;
 4    STACK stream_stack = new_stack();
 5
 6    while(1)
 7      {
 8        while( code_stream -> code<new_stream>)
 9          {
10            if( new_stream)
11              {
12                push( stream_stack, code_stream);
13                code_stream = new_stream;
14              }
15            else
16              { print_code( code);
17              }
18          }
19        if( empty( stream_stack))
20          break;
21        else
22          code_stream = pop( stream_stack);
23    }
24  }
```

第 5 章

意味処理用仮想マシン

ストリームに基づいた意味処理の並列実行モデルに基づいた並列コンパイラの最初期の実現 [Nishiyama 90a-b] では，SunOS 上で Light Weight Process[Sun 91] を時分割処理することによって，仮想的なプロセッサ上で意味解析処理を行なうプロセスの切り替えを行い，プロセスの実行状態の変化からプロセッサ稼働率を予測していた．この手法によって意味処理の並列実行モデルの実現可能性を検証することはできたが，コンパイラの意味処理部の高速な実現やハードウェアによるサポートなどを考えるためには実際のハードウェアシステム上での意味処理実行時の動的な特性の精密な評価が不可欠であると考えた．そこで，この問題に対応するためのマルチプロセッサアーキテクチャ SMiS を設計し．その命令レベルのシミュレータを開発して，クロック単位でのプロセスの実行時間を測定することにした．さらに，並列意味処理そのものと，プロセス間通信やプロセス管理のために必要な処理を分離して測定することを容易にするために，後者をプロセスの‘実行環境’である仮想マシンとして抽象化し，その実現を行なった．2 章で述べたように，ストリームに基づいた意味記述モデルでは，入力プログラムに対応して意味処理を行なうプロセスを動的に生成し，各プロセスがストリームを介して通信を行なう．このため，並列処理用の仮想マシンではこれに対処する機構を用意している．Transputer[Inmos 86] 上で実行される Occam などにみられるプロセスの生成やチャネルによる同期通信のための機構のように，これらの機能を含めて全てをハードウェア化することは不可能ではないが，ここでは，特殊なハードウェアのメカニズムは用いないで，ごく一般的なハードウェアの構成を前提とすることにしたので，特殊な機能はソフトウェアで実現することとした．

5.1 仮想マシンプリミティブ

本節では，仮想マシンが意味解析処理を行なうプロセスに提供するプリミティブに関して説明を行なう．仮想マシンプリミティブは，意味処理を行なうプロセスが要求するプロセスの動的な生成，ストリームを介したプロセス間通信などを実現する．これらのプリミティブは C 言語のライブラリとしてハードウェア上に実現されており，SSDL によるプロセスの定義や C 言語によるプロセス定義から呼び出されることによっ

て利用される．

(1) プロセス生成プリミティブ

仮想マシンは意味処理を行なうプロセスに対してメモリ，CPU，プロセス間の通信機構などの実行環境を提供するものであり，1つのプロセスに対して1つの仮想マシンが対応して存在するものと考える．プロセス側からは各仮想マシンは独立して動作するように見え，意味処理部ではスケジューリングや実際のプロセッサ数などの実現の詳細に関しては考慮する必要はない．

仮想マシンにはプロセスの生成と消滅を行うために，次の2つのプリミティブがある．

- `create_process(proc, arg, ...)`
  新たにプロセスと仮想マシンの組を生成し，proc で指定した手続きを仮想マシン上で実行する．

- `delete_process()`
  このプリミティブを実行中のプロセスと仮想マシンを消す．

このプロセス生成のためのプリミティブでは，UNIX の fork などの場合とは異り，生成するプロセスと生成されたプロセスとの間で環境の継承などは行なわないが，プロセスの生成時に引数としてストリームを与えることにより仮想マシンの間を通信路を確立することができる．プロセスは仮想マシンに接続されているポートにアクセスすることにより，プロセス間のストリームによる接続は仮想マシン間で自動的に行われる．

(2) メモリ管理プリミティブ

各プロセスに対応した仮想マシンは全仮想マシンで共有されるメモリと，各仮想マシンに固有なメモリを持つ．共有メモリ空間は全ての仮想マシンで同一のアドレスに存在する．メモリ管理のためのプリミティブとしては，共有メモリと局所メモリのそれぞれに対する確保と解放を行うため，次の4つのプリミティブを用意する．確保された共有メモリは，全ての仮想マシンから同一アドレスでアクセスすることが可能であり，プロセス間のポインタの受渡しにより情報の共有が可能となる．ただし，ポインタによる情報の共有を行なう場合には，プロセス間の共有データの一貫性をプログラムの記述時に保証するように留意する必要がある．SSDL 等によるプロセスの記述では，基本的にデータの書き換えは必ず1つのプロセスで行ない，ポインタの送信によるデータの共有後はデータの共有を行なわないことによってこれを保証している．

- `malloc(size)`
  size で指定された大きさの領域を局所メモリに確保し，そのアドレスを返す．

・ mfree(address)
局所メモリ中に確保されている address で指定した領域を解放する.

・ shmalloc(size)
size で指定された大きさの領域を共有メモリに確保し，そのアドレスを返す.

・ shmfree(address)
共有メモリ中に確保されている address で指定した領域を解放する.

この共有メモリは，意味処理を行なう各プロセス間で，記号表内のデータや中間コードなど，ストリームによって受け渡されるデータの共有を効率良く行なうために使用し，仮想マシンに固有なメモリはプログラム・テキスト，スタック，局所データなどを格納するために使用する．共有メモリをプロセス間の低レベルの通信機構とすることも可能であるが，同期やアクセスの制御が煩雑になるため，仮想マシンのレベルでは共有メモリはデータの共有機構としてのみ用い，プロセス間の通信と同期のための機構としては，次に説明するストリームを用意している.

(3) プロセス間通信プリミティブ

仮想マシンレベルでのプロセス間の通信機構としてはストリームによる同一の通信路を介した通信機能を提供する．ここでのストリームは単方向の通信を可能とする抽象データであり，プロセス間のデータの転送や通信路の結合は仮想マシンのプリミティブより行なわれる．ストリームは以下に示す仮想マシンプリミティブにより動的に生成される．生成されたストリームはプロセス生成時に手続きの引数として渡され，プロセス間を結合する.

・ create_stream()
新たなストリームを生成し，そのストリームの識別子を返す.

まず，1対1のプロセス間の通信に用いられるストリームへの送信と受信，及びストリーム中のデータの存在を確認するための操作を示す．1対1の通信は主に中間コードの受け渡しなど1方向のプロセス間通信に使用する．ただし，Occam に見られるようなデータのコピーを行なうような通信はデータの大きさが1ワードに収まるものだけに制限し，より大きなデータについては共有メモリ上のデータの格納アドレスのみを受け渡す．このためのプリミティブは次の3つである.

・ send_stream(stream, data)
stream で指定したストリームを通して data で指定したデータを送る.

・ recv_stream(stream, address)
stream で指定したストリームからデータを受け取り，address で指定され

た領域に格納する．データの受信が行なわれれば真を返し，ストリームが閉じられていれば偽が返される．

・ poll_stream(stream)
stream で指定したストリームにデータがあるかどうかを確認する．ストリームが閉じられていれば偽を返す．

2つのストリームを組にして複数のプロセスで共有することにより，多対1の通信を行うことも可能である．ストリームによる多対1の通信は意味処理のレベルでクライアント・サーバ型の通信を実現するためのもので，複数のプロセスが同一のストリームを介して1つのプロセスに要求を送り，結果を受けとることを可能にする．この多対1の通信は意味処理部のレベルでは記号表の管理などに使用する．このための仮想マシンプリミティブとして，サーバへの要求の送信と結果の受信をアトミックに行う操作を提供する．

・ send_and_recv_stream(stream1, stream2, data)
stream1 で指定したストリームを通してサーバ・プロセスに data で指定した要求を送り，サーバからの結果を stream2 から受けとる．

サーバ側では，recv_stream によりクライアントからの要求を待ち，send_stream により結果を返す．クライアントは send_and_recv_stream によりサーバに要求を送り，結果を受け取る．send_and_recv_stream はアトミックなプリミティブであるため，複数の要求があった場合には排他制御が行われる．このため，send_and_recv_stream を用いた場合には，2つのストリームは実質的には大きさ0の双方向のキューと同じ働きをする．

ストリーム自身をストリームを介して受け渡すことによって，通信路の構成を動的に変更することが可能である．ストリーム中のデータがストリームかどうかは，次に示す操作によって確認することができる．この操作により，ストリームをストリームのデータとして受け渡す事ができるようになり，通信路の構成を動的に変更することが可能になる．

・ is_stream(data)
data で指定されたデータがストリームかどうかを確認する．

5.2 マルチプロセッサシステム SMiS

前節で述べた仮想マシンのハードウェア部の実現としてマルチプロセッサのハードウェアアーキテクチャ SMiS の設計を行なった．SMiS で想定しているハードウェアは図 5.1 に示すように，複数台のプロセッシング・エレメントと共有メモリからなる密結合システムである．各プロセッシング・エレメントは SPARC CPU とローカルメモリ，キャッシュから構成され，図に矢印でしめすように SPARC CPU とローカルメモリが接続され，また，キャッシュと共有バスを介して共有メモリに接続されている．

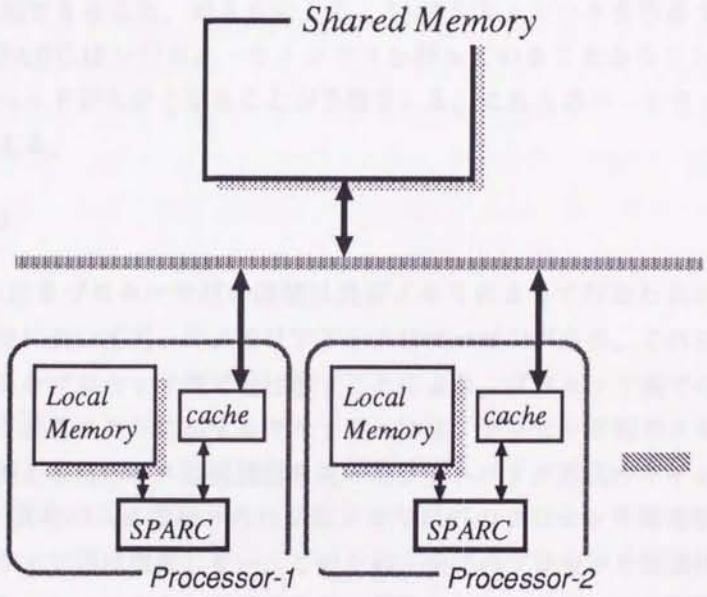

Shared Memory
Local Memory
cache
SPARC
Processor-1
Local Memory
cache
SPARC
Processor-2

図 5.1 SMiS の構成

(1) CPU

SPARC は RISC 型のアーキテクチャ [Katevenis 84] を持つ CPU であり，単純かつ少数の命令セット，親子関係にある手続き間でのレジスタのオーバラップを許すレジスタ・ウィンドウ，マルチプロセッサシステムをサポートする命令，などの特徴を持つ． SPARC は幾つかの実現が行なわれているが，ここでは Cypress 社の CY7C600 シリーズ [Cypress 90] による実現を仮定している．また，ここでの利用では浮動少数点計算および仮想メモリは必要としないので，浮動少数点ユニットおよび，メモリ管理ユニットはハードウェアに付加しない．

SMiS の CPU として SPARC を採用したのは，1)RISC CPU であることから命令数が少なく単純なため，シミュレータの開発が容易であり，2) マルチプロセッサ・システムに対応した命令を備えていること，3) 現在使用しているワークステーションの開発環境が流用できること，の 3 点による．1) はシミュレータを作成する上では利点となるが，SPARC はレジスタ・ウィンドウを持っていることからコンテクスト切替時のオーバーヘッドが大きくなることが予想される．これらのハードウェアに関する評価は 6 章で与える．

(2) 共有メモリ

SMiS におけるプロセッサ間の通信は共有メモリによって行なわれる．共有メモリは各プロセッサにおいて同一のメモリアドレスにマッピングされ，これにより共有メモリ上のアドレスをプロセッサ間で受け渡すことにより，プロセッサ間でのデータの共有が実現される．共有メモリに対するキャッシュには，プロセッサ間のメモリ・アクセスの競合を低く押えるためバス監視機構を用いたライトバック方式のキャッシュ [Archibald 86] を仮定し，共有バスに接続された共有メモリ以外のプロセッサ間通信機構としては特殊なハードウェア等は用意しない．このため，全てのプロセッサ間通信は，共有メモリ上でのオペレーションにより実現される．具体的には，プロセッサ間通信は不可分メモリアクセス命令 swap および ldstub を用いて低レベルのロック・オペレーションを実現し，これを組み合わせてストリームなどの高レベルの通信および同期機構を構成する．ここで，swap 命令は指定したレジスタとメモリ上のワードを交換する命令，ldstub はメモリ上の指定したバイトの値を指定したレジスタに読み込むと共に当該バイトの値を 0xFF とする命令である．これらの命令は，CPU アーキテクチャの定義によって複数のプロセッサで同時に実行した場合に排他的な実行が行なわれることが保証されている．ローカルメモリおよび共有メモリは CPU との間にキャッシュを持ち，キャッシュ・ヒット時のメモリ・アクセスは 1 クロック，ミス時のメモリアクセスには 6 クロックを要するものとする．

(3) プログラム実行環境

図 5.2 に SMiS アーキテクチャにおける各プロセッサのメモリ・マップを示す．SMiS のメモリマップは基本的に UNIX の標準的なメモリモデルに従って構成されている．

従ってメモリの下位番地にコードと静的データを配置し，その後に動的に確保されるヒープを置き，上位番地にスタックを配置する．ヒープはメモリの上位方向に増加し，スタックは下位方向に伸びる．プロセッサ間の共有メモリはヒープとスタックの中間に配置される．また，メモリ空間の最下位にはトラップ処理用のデータ等が置かれ，最上位にはシステム構成に関する情報を格納する．具体的には，メモリの 0x00000000 から 0x00001FFF には割り込みベクタとその処理ルーチンを格納する．さらに，0x0000 2000 から 0x7FFFFFFF まではプログラムのテキスト，および静的なデータに割り当て，0x8F000000 以降はスタックとして使用する．共有メモリには 0x80000000 から一定の大きさの領域を割り当てる．また，0xFFFFFFF0 から 0xFFFFFFFF までには，レジスタ・ウィンドウの大きさやプロセッサの台数など，実行環境に関する情報を格納する．

共有メモリの最初のワード（アドレス 0x80000000）は実行の開始時に 0xFFFFFFFF に初期化される．これは，システムが実行を開始した時点でシステム全般に渡る初期化を，特定のプロセッサで行なうために用意されている．システムが実行を開始すると，初期化を行なうプロセッサは大域的な初期化処理を開始する．これを除いた他のプロセッサは，当該ワードが 0xFFFFFFFF の間初期化の終了を待つ．初期化処理を行なうプロセッサは，初期化終了後当該ワードの値を 0 とし，他のプロセッサに初期化の終了を知らせる．

5.3 スレッドライブラリ

意味処理実行のためのハードウェア・システムである SMiS と意味処理プロセスとの間の仮想マシンを実現するソフトウェア部は，軽量プロセスであるスレッドの動的生成とその通信機構などを提供するスレッドライブラリとして C 言語により実現されている．このスレッドライブラリは手続き呼び出しの形式で意味処理部から呼び出され，ハードウェアで直接実現することの難しい仮想マシンの機能の一部を実現している．現在の所，スレッドライブラリが実現しているのは仮想マシンのプロセス生成とストリームによる通信機構，および動的メモリ割り当ての機構である．SMiS のハードウェアのレベルではプロセッサの間の通信機構としては共有メモリしか存在しないため仮想マシンの管理情報の一部とストリームは共有メモリ上に実現している．

(1) プロセス管理方式

プロセスはプロセス生成を行なうプリミティブによって動的に生成されるため，実際の実現では各プロセッサ上で，その実行母体である複数の仮想マシンを動的に切替えながら実行を行なう方式を取っている．この仮想マシンの管理には基本的には FIFO 型のスケジューリング方式を用いている．

プロセスとこれに対応する仮想マシンの生成は，共有メモリ上のプロセス生成キューに生成されるプロセスの情報を置くことによって行なわれる．ここでの，プロセス生成のための情報はプロセスとして実行される手続きのアドレス，この手続きの初期パラ

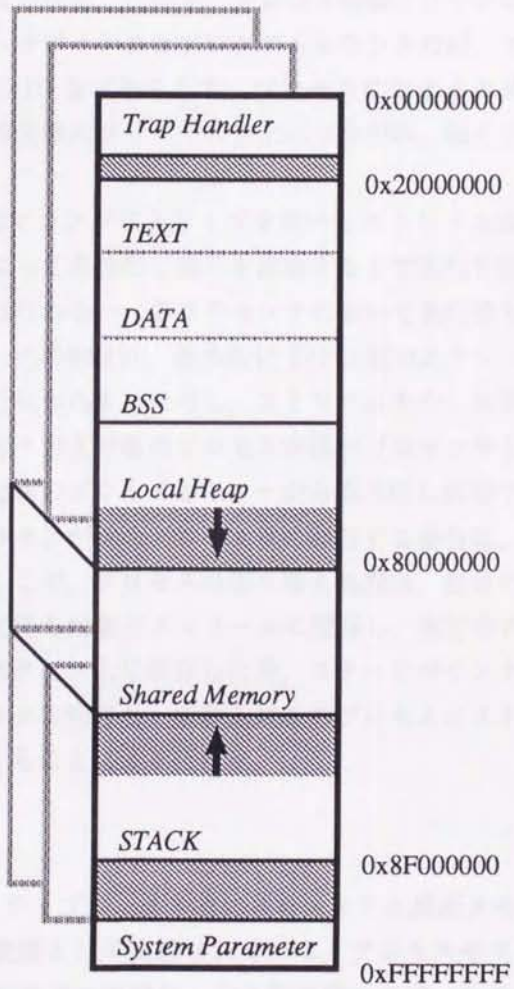


図 5.2 SMiS のメモリマップ

メータ等からなる．各プロセッサは，アイドル状態になるとこのキューから情報を取り出し，プロセスの生成を行なう．プロセスの生成は，(a) プロセス実行のためのスタック，プロセス制御ブロックなどプロセスの実行を行なうために必要な領域の局所メモリおよび共有メモリ上への割り当て，(b) プロセスの初期化を行なう手続きの起動，(c) プロセスの生成に関する情報で指定された手続きの指定されたパラメータによる，プロセス初期化手続きからの呼びだし，によって行なわれる．プロセスの実体に対応する手続きの実行が終了すると，プロセスの初期化を行なう手続きへ制御が戻り，プロセスの消去などの処理が行なわれる．ここで，プロセス制御ブロックは，プロセスの停止時の環境に対応するスタックポインタとプログラムカウンタの対，プロセスの実行状態，実行されるプロセッサの ID などからなる．プロセスに対するスタックとプロセス制御ブロックはプロセスの消去時にフリーリスト上につながれ，続くプロセス生成時に最利用される．

各プロセスは仮想マシンプリミティブを用いたストリーム通信による通信待ちか，プロセスの終了等によって自発的に実行を放棄するまで実行を続け，時分割処理等による強制的な切り替えは行わない．各プロセッサにおいて実行待ち状態にあるプロセスは共有メモリ上のキューに格納され，基本的に FIFO 型のスケジューリングによりプロセスの切り替え処理が行なわれる．ただし，ストリームを介した通信によってプロセスの切り替が行なわれ，切り替え対象のプロセスが同一プロセッサ上で実行待ち状態にある場合には，切り替え対象のプロセスをキューから取り出し直接プロセスの制御を切替える．切替え対象のプロセスが他のプロセッサに存在する場合は，切替え対象のプロセスを待ち状態に置く．ここで，プロセスの切り替え処理は，自身のプロセス制御ブロックを待ち状態に設定して待ち対象のストリームに登録し，実行中の CPU 内の有効なレジスタ・ウィンドウをスタック上に保存した後，スタックポインタとプログラムカウンタをプロセス制御ブロックに保存し，切替え対象のプロセスのスタックポインタとプログラムカウンタに切替えることにより行なう．

(2) メモリ管理方式

メモリ管理プリミティブは，基本的に共有メモリと局所メモリに対する可変長の動的なメモリ割り当て機構として実現されている．プロセスやストリームに対するメモリや，コンパイラの記述での中間コードや記号表内のデータなどの記憶される動的な領域はこのメモリ管理プリミティブを介して割り当てられることになる．共有メモリに対応したメモリ領域のうち，プロセス生成，管理用のキューなど全プロセッサからアクセスされる大域的なデータ領域を除いたメモリは，プログラム全体の実行の開始時にプロセッサ毎に管理される領域に分割される．各プロセッサではこの領域を要求に応じて分割することによって独自にメモリ割り当て操作を行なうことにより，共有メモリの動的な割り当て管理を行なう．同様に，局所メモリの割り当てに関しても，静的データの次に割り当てられたヒープ領域を要求に応じて細分することにより，メモリ割当を行なう．これらのメモリ割り当てを行なうための空き領域に対するポインタや，空きメモリのサイズなどは局所メモリ上の静的領域上に確保される．メモリ管理プリミティブは，

プロセスの生成や共有データの構成を行なうために頻繁に呼び出される．初期の実現では共有メモリ上の管理領域を複数のプロセッサから操作することによって共有メモリの動的割り当て処理を行なっていたが，これは空き領域管理のためのデータ操作を行なうために領域のロック操作が必要であり，メモリ割り当て操作の競合による性能の低下を招いていた．現在の実現では，共有メモリ領域を起動時にプロセッサ毎に分割することによって，メモリ管理情報のロック操作を不要とし，メモリ割当のための管理情報操作の競合を緩和している．

(3) ストリームの実装方式

ストリームによる通信機構は，基本的に固定長の環状バッファとして共有メモリ上に実現される．現在の実現でのバッファ長はデフォルトで 10 である．この環状バッファは，データを格納するバッファおよびデータの先頭および末尾を示すポインタと，ストリームの状態を示すフラグから構成される．バッファの各要素は 1 ワードのデータと附属したタグによって構成され，このタグはバッファの要素がストリームかどうかの判別に使用される．また，各ストリームにはプロセッサ間での排他制御を行なうためのロック領域と，生成者，消費者，待ち状態のプロセスそれぞれに対応するプロセス制御ブロックのポインタが附属している．ストリームに対する送信や受信などを行なうプリミティブでは，プロセッサ間での操作の競合を排除するため，共有メモリ上で SPARC の不可分メモリアクセス命令 swap および ldstub を用いてストリームをロックした後，操作を行なう．アトミックな操作に対しては，ストリームは 5.1 節で述べたように大きさ 0 のキューとして扱われる．

ストリームへの送受信を行なうプリミティブにおいて，バッファが一杯，あるいは空であるという理由によって送信あるいは受信ができない場合には，(1) で述べたプロセスを管理するモジュールが呼び出されプロセスの切り替が行なわれる．この際，自身のプロセス管理ブロックへのポインタをストリーム中の待ち状態リストに登録する．同一プロセッサ上で実行されているプロセスへのデータの送信あるいは受信によってプロセスの切替が発生した場合，ここから待ち状態にあるプロセスを取り出すことによって (1) で述べたように可能な限りそのプロセスへ直接実行の制御が移される．これにより，複数の動的なプロセスの実行と通信によって生じる，無駄なプロセス切替が極力減少されるようにしている．

データベースシステムのように大量のデータをストリーム処理によって扱う場合には，ストリームに対するバッファの割り当ての最適化が問題となる場合があるが [Liu 88]，ここで対象としているような言語処理系などでは，扱うデータは高々生成されるコード程度であり，このような処理は特に必要がないと考えている．

5.4 SMiS シミュレータ

SMiS シミュレータはを UNIX 上で SMiS アーキテクチャのシミュレーションを行なうためのソフトウェアシステムである．このシミュレータは，当初ハードウェア記述

言語 ISP'[ZYCAD 88] によって記述されていたが，実行の高速化と実行プログラムの処理の柔軟性を実現するため，C 言語に書き直された．

前節で述べたように，SMiS の各プロセッサにおけるメモリ空間の利用は，UNIX のプロセスのメモリモデルにほぼ従っている．加えて，SMiS シミュレータは UNIX 上でコンパイル / リンクした実行ファイルをシミュレータ上に読み込むことを可能にしている．これにより，SMiS 上のプログラムの開発に，UNIX 上の開発環境を流用することを可能としている．ただし，現在 Sun4 上に実現されている SMiS シミュレータは，いくつかのシステムコールと共有ライブラリ [Sun 91] をサポートしていないため，これらを利用したプログラムは動作しない．

このシミュレータでは命令の基本実行クロックを Cypress の CY7C600 シリーズに合わせており，大部分の命令は内部パイプラインにより実質 1 クロックで実行されるよう設計されている．ただし，シミュレータでは内部パイプラインの実現は行っていない．また，対象としているアプリケーションでは浮動小数点演算は行なわないので，浮動小数点ユニットは実現していない．メモリアクセスに関しては，PE のローカル / 共有の各メモリは CPU との間にキャッシュを持ち，キャッシュ・ヒット時のメモリ・アクセスは 1 クロック，ミス時のメモリアクセスには 6 クロックを要すると仮定している．現在のシミュレータでは，このキャッシュのヒット率を変化させることでキャッシュの効果をシミュレートしており，キャッシュミス時に共有メモリへのアクセス競合が起こった場合，メモリアクセスに対してクロックディレイが挿入される．SMiS シミュレータは，プログラムの実行開始時，全プロセッサに同一のメモリイメージを読み込み，各プロセッサはメモリ空間の最上位に格納されたプロセッサ ID によってプロセッサ毎の処理を選択する．

SPARC アーキテクチャでは，プロセッサ内のレジスタ・ウィンドウの数はハードウェアの実現依存となっているが，SMiS シミュレータではレジスタ・ウィンドウの段数はオプションによってシミュレーションの開始時に指定することが可能である．また，SMiS シミュレータでは，実行対象のプログラムの詳細な実行データを得るために，各プロセッサにおける手続きの呼び出し回数や，総実行クロック数などの実行プロファイルを得る機能も実現している．

シミュレータは，UNIX 上の 1 つのプロセスとして実現され，各プロセッサに対応して実行のスレッドとレジスタやメモリ等の環境を，独自の軽量プロセス・ライブラリによって実現している．各プロセッサの実行は，プロセッサに対応する軽量プロセスを 1 クロック毎に切替えることにより行なわれる．現在実現されている SMiS シミュレータの実行速度は，プロセッサ数を 1 台とした場合，約 0.004MIPS であり，シミュレーションに用いたハードウェア (SparcStation 1) の約 2800 分の 1 の速度で実行を行なう．

第 6 章

細粒度プロセスによる実行方式

細粒度プロセスによる並列意味処理方式は，ストリームに基づいた並列意味解析モデルを素直な形で SMiS 上に実現したものである．ここで対象とする意味処理のモデルでは，意味処理を行なうプロセスは入力プログラムに対応する解析木のノードに対応して生成される．細粒度プロセスによる実現では，中間的な解析木の生成は行なわず，プロセスの生成とストリームによる結合処理は構文解析と同時に行なう．ただし，ストリーム間を受け渡されるデータの依存関係からの制約により，その実行が逐次的な評価と同等なものとなる場合がある．細粒度プロセスによる実現では，これに対処するため，SSGL による文法および意味規則の定義時にプロセスの属性を指定することにより，構文の認識時にプロセスの評価順序に対応した評価グラフを構成し，これを辿りながら評価することでプロセスの生成を抑制する機構を用意している．この実現では，構文解析器として LR 解析法を用いたボトムアップ型の解析法を仮定し，構文の認識に対応した構文解析動作を，制御プロセスと呼ばれるプロセス生成処理を司るプロセスが受け取りプロセスの生成処理を行なうという方式を採っている．本章では，この実行方式とその実現，SMiS によるシミュレーションに基づいた評価を示す．

6.1 実行方式

細粒度プロセスによる並列実行のモデルは，基本的には 2 章で述べたストリームに基づいた意味記述モデルの直接的な実現であり，基本的に解析木上の各ノードに対応してその意味処理を行なうプロセスの割り当てを行なう．ただし 2 章のモデルをそのまま実現したのでは，プロセスの生成は行なわれるものの，ストリーム中のデータの依存関係によっては，ある一意の順序でのプロセスの実行しか選択できない場合がある．現在の一般的なアーキテクチャではプロセスの実現には大きなコストを要することが通例であり，性能面からは好ましくない．ただし，SSGL 記述はプロセスの結合関係を記述するものであり，ストリームがどのように参照されるかという情報はここからは得られない．このため，3 章で述べたように SSGL 記述ではプロセス名の前に 'SEQ' を前置することにより，プログラマが言語定義を行なう際にプロセスに対して逐次化の指定を行なうことができる．逐次化対象のプロセスは，仮想的なプロセスとして扱われ，実行

時に解析木上で隣合うものがまとめられ，実際には1つのプロセスとして同一の実行コンテクストを用いて実行が行なわれる．

6.1.1 実行方式の概要

細粒度の実行モデルにおいては，図6.1に示すように，意味解析器内の制御プロセスと呼ばれるプロセスが構文解析に関する情報を構文解析器から受けとり，認識された解析木のノードに対するプロセスの生成と，プロセス間のストリームによる結合を行なう．ここで，逐次化対象ではないプロセスに関しては，その生成は対応する非終端記号の認識に対応した還元動作の受理によって行なわれる．プロセス間を結合するストリームに関しては，構文解析用の状態スタックと同期して動作する制御プロセス内の属性スタックを介して受渡しが行なわれる．逐次化対象のプロセスに対しては，制御プロセス上でボトムアップの構文解析に対応して逐次評価グラフと呼ばれる構造が生成され，これを逐次的に辿りながら値の評価を行なうプロセスの生成が行なわれる．逐次評価グラフは，複数のプロセスによって共有される場合があるため，その評価時には複数のプロセス間で競合が起こらないように調整が行なわれる．なお，図6.1では，構文解析部によって認識された解析木の構造を破線で示し，プロセスを白抜きの丸で示している．また，構文解析部から送られる構文解析動作に対応した制御プロセスによるプロセスの生成過程を矢印で示している．

6.1.2 制御プロセス

制御プロセスは，構文解析部から入力テキストの構文解析を行なった結果を受け取り，実際の意味解析処理を行なうプロセスの生成を行なう．制御プロセスは，プロセス間の結合に使用されるストリームや逐次評価グラフを生成規則間で受渡すために使用される属性スタックを内部に持ち，このスタックは構文解析器における解析状態スタックと同期して操作される．意味解析プロセスの生成は，基本的に次に示すようなアルゴリズムで行なわれる．

```
Step1:
  構文解析動作を読み込む;
  if( (構文解析動作 == 還元) &&
      (意味規則が定義されている))
       goto Step2;
  else goto Step1;

Step2:
  if( 対応するプロセス == 逐次プロセス) goto Step3;
  else if( ( プロセス == ストリームのコピー) &&
           ( 被コピー側のストリーム == 既出))
    {
      被代入側のストリームを代入側のストリームと同一にする;
      goto Step1;
    }
  else
    {
```

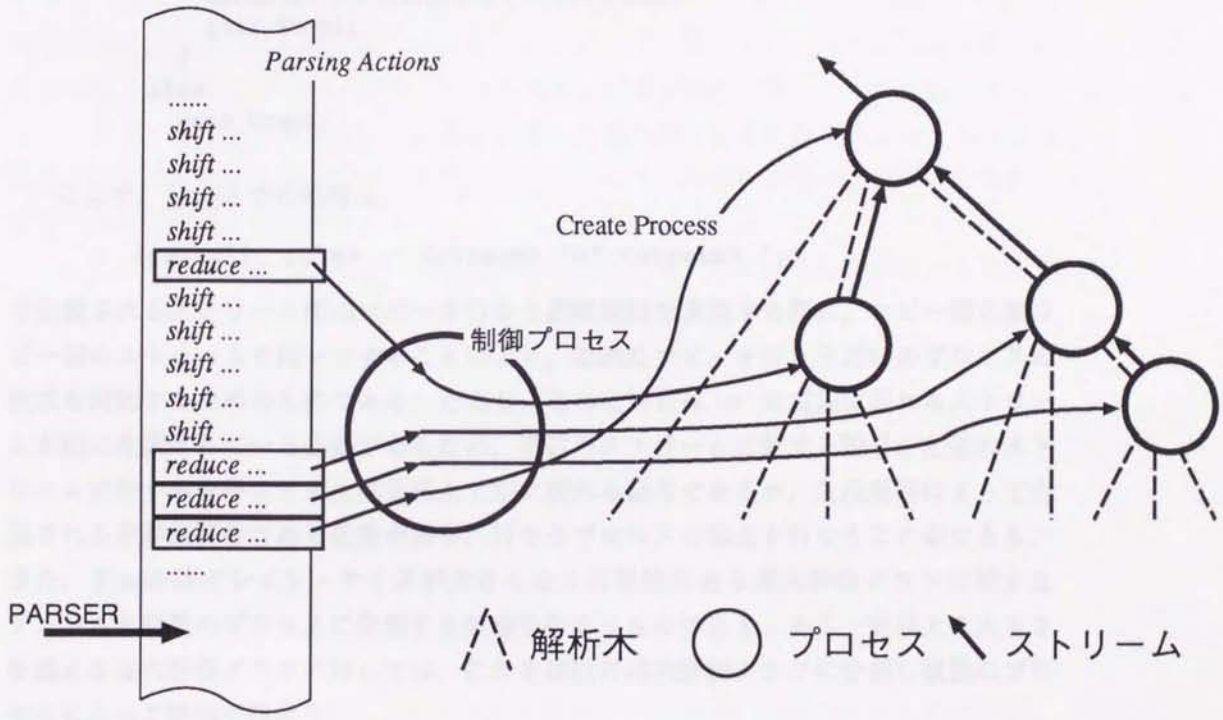


図 6.1 細粒度プロセスによるプロセス生成

```
            プロセスを生成．プロセスの引数に逐次プロセスがあれば，こ
            の評価をプロセス本体の実行前に行ない，結果をプロセスの引
    数とする;
            goto Step1;
          }

    Step3:
      if( 次の還元動作に対応するプロセス == 並列プロセス)
        {
          逐次評価グラフを生成;
          goto Step1;
        }
      else if( 逐次評価グラフの大きさ >= α)
        { /* ここで，αは一定の値 */
          逐次評価グラフを評価するプロセスを生成;
          goto Step1;
        }
      else
        goto Step1;
```

ここで，Step2での処理は，

```
<semantic rule> → <stream> '=' <stream> ';'
```

で定義されるストリーム間のコピーを行なう意味規則を実現する際に，コピー側と被コピー側のストリームを同一とすることにより，単純なコピーを行なうだけのプロセスの生成を抑制するためのものである．ただし，このためには'='の右辺に現れるストリームが既に生成されている必要があるため，右辺のストリームに対する記号が左辺のストリームに対する記号よりも生成規則上で左に現れる記号であるか，生成規則によって定義される非終端記号である必要がある．行なうプロセスの除去を行なうことができる．また，Step3はグレイン・サイズが大きくなる可能性のある逐次評価グラフに対するプロセスを複数のプロセスに分割する処理を行なうものである．ある一定以上の大きさを越える逐次評価グラフに対しては，これを複数の逐次評価グラフに分割し複数のプロセスによって評価を行なう．

6.1.3 プロセスの逐次化

上に述べたように，意味処理を並列プロセスとして実行する場合には，プロセス間通信のコストやプロセス切り変えのオーバーヘッド等の問題から，概念的には並列実行を行なうことの可能なプロセスについても，それを複数のプロセスではなく，同一の実行コンテクスト上で1つのプロセスとして実行する方が実行の効率が良い場合がある．このため，細粒度の実現モデルでは，解析木上で連続する複数の仮想的なプロセスを1つにまとめて同一の実行コンテクストにより実行する逐次プロセスと，独立に実行されるプロセスとが存在する．逐次プロセスの実行は，複数の逐次プロセス間でプロセスの実行環境を構成する仮想マシンが共有され，1つのプロセスとして実行される仮想的なプロセス群の情報の受渡しはスタックを介して行なわれ，その値は列的な構造を持たな

い値として扱われる．このため，プロセス間通信やプロセス切り変えに要する手間が不要となる．

プロセスの逐次化は，逐次評価グラフと呼ばれる構造を用いて行なわれる．逐次評価グラフの各ノードは，仮想的なプロセスとして実行される関数とその引数へのポインタから構成され，この他に評価結果を格納するフィールド，データが評価されたかどうかを示すフラグと，そのデータを計算しているプロセスの id を格納するためのフィールドが付加される．このグラフを深さ優先に辿り，ボトムアップに全てのノードにおける関数を呼び出し，結果の値をノードに格納することによって評価が行なわれる．

この逐次評価グラフは，解析木上で連続した逐次プロセスに対してその評価順序に対応するプロセスの評価グラフを構文解析と同時に行なうことによって構成される．このため，あるプロセスを逐次プロセスとして指定する場合には，属性文法における L-属性 [Bochmann 76] と同様に，解析木のあるノードに属したストリーム定義の代入の右辺に現れるストリームが，そのノードの親あるいは左側の兄弟のノードに属したストリームでなくてはならないという制限がある．生成された逐次評価グラフは，その結果を必要とする逐次化指定を行なわれていないプロセスの生成前に値の評価が行なわれ，プロセスの生成時に値が渡される．

ここで，例として，中間コード生成処理を行なう際，式に対応したプロセスに対して逐次化の指定を行なった場合を考える．この場合，式に対応するプロセスのネットワークに対応したグラフが生成され，その値を必要とする逐次化指定を行なわれないプロセスの生成前にその評価が行なわれる．ここで，図 6.2 に示すように，入力として while 文が与えられ，while 文に対応したプロセスに関しては逐次化指定が行なわれていない場合を考える．ここでは，解析木の部分的な構造を破線で示し，while 文の認識に対して生成されるプロセスを白抜きの丸で示している．また，while 文の条件部に対応して生成される逐次評価グラフを破線で囲んで示している．while 文の条件式に対応した解析木の認識に対して，図に斜線で囲んで示すように exp_ge(exp_add(exp_var(), exp_val()), exp_var()) という関数呼びだしに対応した逐次評価グラフが構成される．while 文に対応する処理を行なうプロセスに対しては逐次化の指定は行なわれていないため，while 文の条件部の式の構造に対応して作成された逐次評価グラフを辿ることによって評価が行なわれ，その結果が while 文に対応するプロセスに受け渡され，while 文に対応するプロセスが実行を開始する．

意味処理の記述を行なう際に，どのプロセスを逐次プロセスとするかについては，生成系によって自動的に判別されることが理想であるが，解析木の構造は入力が与えられるまで得られず，このため SSGL による意味解析処理の記述の静的な情報から，実行時のプロセスの動作を予測するのは困難である．このような理由により，現在の実現では，3 章で示したように SSGL 記述時に明示的に逐次プロセスを指定するという方式を取り，実行時に動的に関数の評価順序に対応した逐次評価グラフを構成することによって動的なプロセスの生成処理を行なっている．

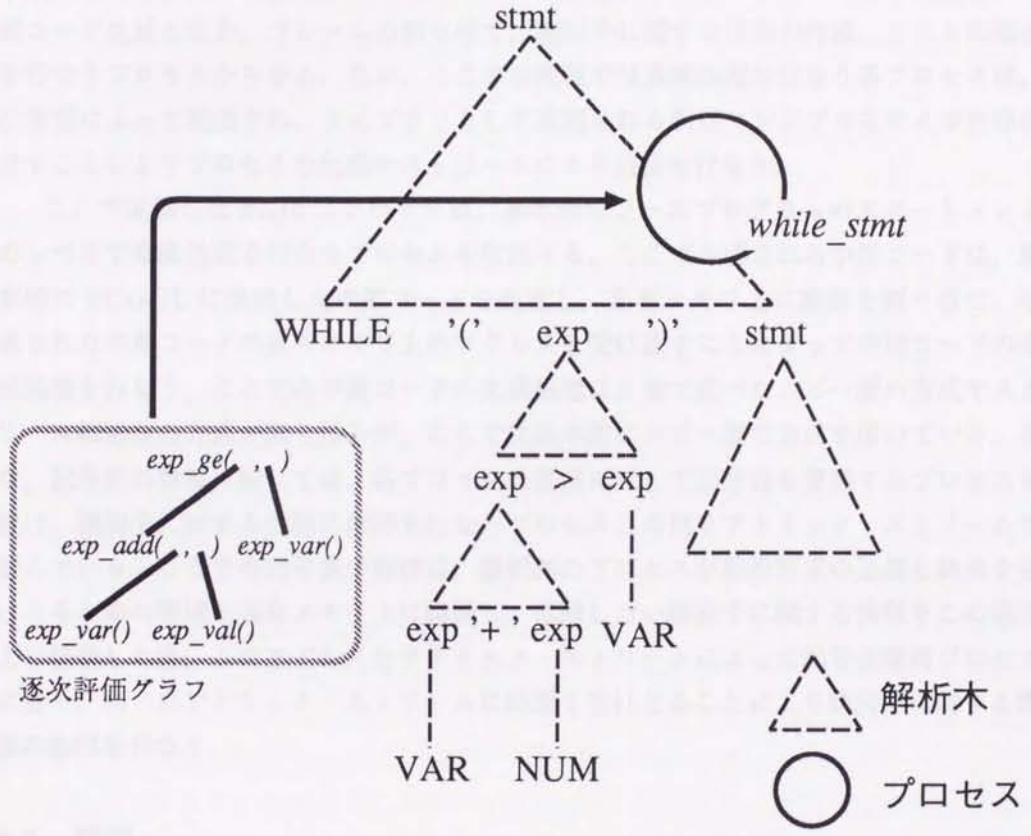


図 6.2 プロセスの逐次化

6.2 PL/0 コンパイラ

本章の評価で用いる PL/0[Wirth 76] コンパイラは，22 の終端記号，21 の非終端記号，46 の生成規則によって文法の定義が与えられる．また，各生成規則に対して生成規則が定義され，その総数は 100 である．意味規則で使用されるストリームは，中間コード，記号表参照，名前情報，名前情報などのリンクに使用されるリスト，フレーム・サイズ，定数値の各々に対応する 6 種である．ここで，ストリームからストリームへの転送を行なうだけの意味規則は 72，プロセスの生成に対応する意味規則は 28 であり，多くは中継を行なう意味規則からなる．ここで，プロセス生成に対応する 28 の意味規則のうち，実際に使用されるプロセスは 16 種からなり，これらは記号表管理，中間コード生成と出力，フレームの割り当て，識別子に関する情報の作成，リストの構成を行なうプロセスからなる．なお，ここでの実現では意味処理を行なう各プロセスは，C 言語によって記述され，ライブラリとして実現される仮想マシンプリミティブを呼び出すことによりプロセスの生成やストリームによる通信を行なう．

ここで実現した PL/0 コンパイラは，基本的にソースプログラムのステートメントのレベルで意味処理を行なうプロセスを生成する．ここで生成される中間コードは，基本的に PCODE に準拠した中間コードを生成し，共有メモリ上に実体を割り当て，生成された中間コードの共有メモリ上のアドレスを受け渡すことによって中間コードの生成処理を行なう．ここでの中間コードの生成処理は 3 章で述べたコピー型の方式やストリーム転送型の方式が取り得るが，ここでは基本的にコピー型の方式を用いている．また，記号表の管理に関しては，各ブロックの宣言に対して記号表を管理するプロセスを設け，識別子に対する情報の参照を行なうプロセスとの間をアトミック・ストリームで結んでいる．ここでの記号表の参照は，参照側のプロセスが参照要求の送信と結果を受けとるための領域を共有メモリ上に確保し，参照したい識別子に関する情報をこの領域上に格納した後，このアドレスをアトミック・ストリームによって記号表管理プロセスに送り，同一のアトミック・ストリームに結果を受けとることにより識別子に関する情報の参照を行なう．

6.3 評価

本節では，細粒度の実現モデルに従って実現した PL/0 の並列コンパイラの評価と，5 章で述べた仮想マシンおよびマルチプロセッサ・アーキテクチャ SMiS の評価を行なう．

6.3.1 細粒度プロセスによる実行性能の評価

まず，テスト・プログラムとして約 250 行の PL/0 プログラムを PL/0 の並列コンパイラによりコンパイルした場合の意味処理部の実行結果を示すグラフを図 6.3 に示す．なお，ここで用いたテストプログラムは 9 つの手続きを持つプログラムであり，SMiS ハードウェアに関しては，キャッシュメモリのヒット率を 100%，レジスタ・ウィンド

ウの段数を 8 としている．また，ここでは参考のため等価な処理を行なう YACC で記述されたコンパイラの実行時間のグラフも同時に示している．YACC によって実現した PL/0 コンパイラは動作ルーチンによる方式であるため，この実現と並列 PL/0 コンパイラは単純に比較可能ではないが，ここでは処理時間の参考のために YACC によるコンパイラの実行時間を示している．

図 6.3 は 3 つの部分に分かれており，上から順に，仮想マシンのスケジューリング等の管理，レジスタ・ウィンドウの管理，意味処理に要した 1 文字あたりのクロック数を表している．横軸はプロセッサの台数に対応している．この結果から，プロセッサ 1 台での実行に要したクロック数とプロセッサ 10 台での実行に要したクロック数を比較すると，5,420 クロックから 560 クロックとなり約 9.6 倍の速度向上となっており，ほぼ，プロセッサの台数に反比例する結果となっている．また，実際の意味処理に使用されている時間は全体の 10% から 20% 程度であり，その他は仮想マシンやレジスタ・ウィンドウの管理を行っている時間である．図 6.3 から分かるように意味処理そのものの時間だけでなく，仮想マシンでの実行時間を含めた全処理時間がプロセッサ数にほぼ反比例している．これは，仮想マシンでのストリーム関連プリミティブやレジスタウインドウの切り替えに要する処理が，複数のプロセッサにそのまま分散されたことを意味している．また，プロセス間の同期待ち等によるオーバヘッドは，現時点では現れていないことも意味している．ただし，仮想マシンでの実行時間が大幅に減少した場合には，顕在化してくる可能性もある．

同様に，意味解析中に動的に生成されている意味処理プロセスのライフタイムを測定した結果意味処理を行なうプロセスのライフタイムの平均は約 50,000 クロックであり，約 10,000 クロックのところにピークがあった．10,000 クロックから 50,000 クロックの比較的短いライフタイムのプロセスは，主に式やステートメントに対応した中間コードの生成を行うプロセスで，生成後短時間で消滅する．これに対して，記号表管理を行なうプロセスや制御プロセス，中間コードの出力を行なうプロセスは，長期間生き続ける．ここでの例では，入力のテストプログラムに対して生成されたプロセス数は 964 であり，同時に存在したプロセス数は，平均 250 程度であった．

また，プロセスが実行のコントロールを得てから失うまでの区間の実行時間を表すランレングスを測定したところ，各プロセスの平均的なプロセス切替の回数は約 22 回であり，平均 1000 クロック程度，ピークは約 700 と約 2000 クロックあたりに見られた．

5 つのサンプルで測定したところ，ソースプログラムの 1 文字当りの意味処理時間には 10-20% の変動が見られたが，プロセスのライフタイムや区間実行時間には大きな変動はなく同じ傾向が見られた．

図 6.3 に示すように，YACC で生成した単一プロセッサ上で動作する逐次型の意味解析器の動作時間は，現在の並列意味解析器のプロセッサ 10 台の時の処理時間の約 1/2 程度であり，並列処理による高速化という観点からすると，まだ，最終的な目標は到達されていない．これは，仮想マシンでの実行時間が大きな割合を占めているためであり，意味処理部の実行時間だけを取り出して比較すると，並列意味処理器の方が 3.6 倍

程度速い．このことから，仮想マシンの効率のよい実現や，プロセス割り当ての最適化が不可欠であることが分かる．

6.3.2 コンパイラの記述方式の評価

ここでは，4章で示したコンパイラの意味処理の実現技法のPL/0コンパイラにおける実現の評価を示す．

(1) プロセスの逐次化

6.1節で述べたように，細粒度の実現モデルにおいては，逐次評価グラフによって逐次的な評価を行なうプロセスを構成することができる．PL/0コンパイラの初期の実現では全ノードに対応するプロセスを並列に実行していたが，シミュレーションによる実行の解析により，プロセス管理など仮想マシン実行のオーバヘッドが高いことが判明したため，式レベルのプロセスおよびステートメントレベルのプロセスに対して逐次化を行ない，プロセスの統合処理を行なった．また，4章で述べたように，コンパイラにおけるコードの生成と受渡しの並列な実現方式としては，中間的な各ノードにおいて下位の構文要素に対応した中間コードを上位のノードに対応するプロセスへ転送する方式と，下位のノードに対応した中間コードに関しては，これに対応した中間コードに対応したストリーム自身を上位のノードにコピーする方式が考えられる．ステートメントレベルでの逐次化では，このような，ストリームを転送する方式によるプロセス生成の抑制も行なわれている．

図6.4にプロセスの逐次化による性能向上に関するグラフを示す．ここで，横軸はプロセッサ台数，縦軸は入力プログラムの1文字あたりの処理に要したクロック数を表している．ここでの入力プログラムには約50行のテストプログラムを用いている．なお，ここで用いたテストプログラムは3つの手続きを持つプログラムであり，SMiSハードウェアに関しては，キャッシュメモリのヒット率を100%，レジスタ・ウィンドウの段数を8としている．ここで，完全並列版は意味規則で定義される全てのプロセスを独立したプロセスとした場合である．式レベルの逐次化は解析木上の式に対応して生成される式の中間コードを生成するプロセスに対して逐次化の指定を行ない，式に対応した中間コードの生成を行なうプロセスのうち解析木上で連続するものを逐次化した場合である．さらに，ステートメントレベルの逐次化では，式レベルの逐次化に加えて中間コードのコピー処理を，ストリーム自身の転送に変換することによってステートメント・リストの処理などに対してプロセスの逐次化を行なっている．図からわかるように，完全並列版と比較すると式レベルの逐次化によって1プロセッサの場合，約4000クロックから約3500クロックとなり約12%の高速化となっている．また，ステートメントレベルの逐次化では，1プロセッサでの1文字あたりの処理時間が約2600となり，完全並列と比較して35%の高速化，式レベルの逐次化と比較して約25%の高速化となっている．さらに，5台のプロセッサで比較すると，完全並列版で約1200クロック，式レベルの逐次化を行なった場合で約1100クロック，ステートメント・レベ

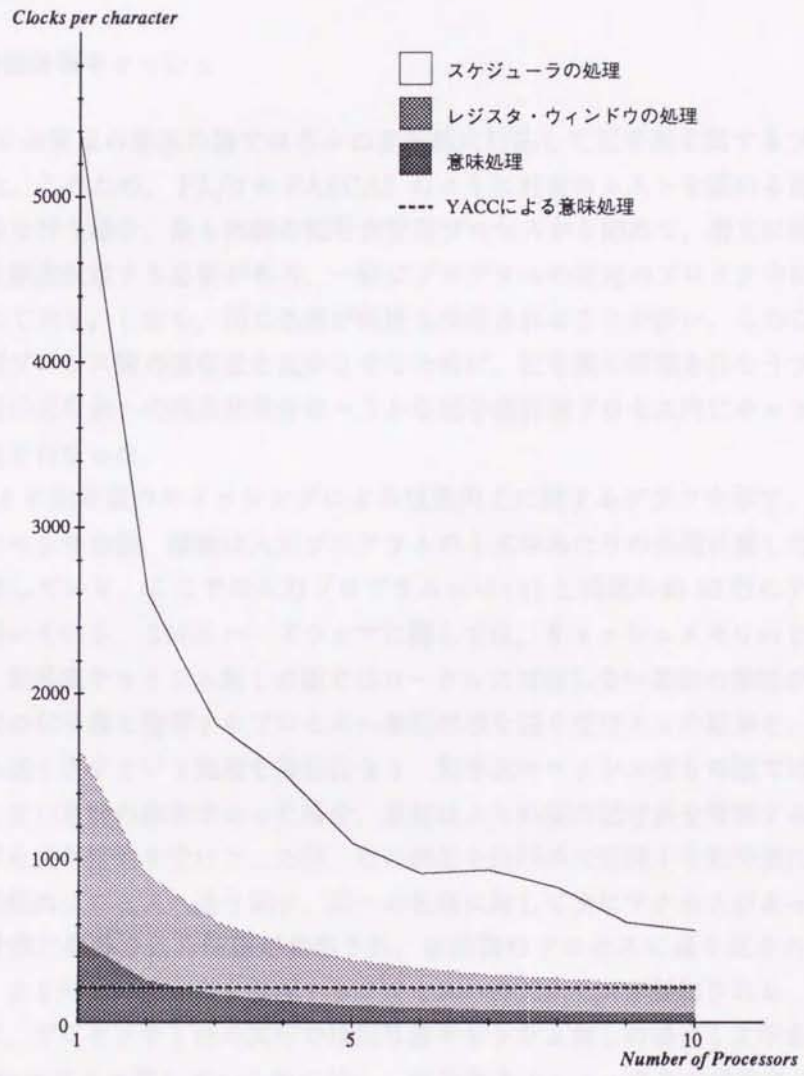


図 6.3 細粒度モデルによる実行結果

ルの逐次化を行なった場合で約900クロックとなり，式レベルの逐次化，ステートメント・レベルの逐次化それぞれに対して，約5%および約25%の高速化となっている．また，式レベルの逐次化では5台以上の処理ではそれほどの高速化は得られていない．なお，ここでは入力されるPL/0のソースプログラムが6.2.1で用いたものと比べて短いことから，プロセッサ数10台の場合に9台での処理の場合と比較して性能の低下が見られる．

(2) 記号表参照キャッシュ

PL/0の実現の最初の版では各々の宣言部に対応して記号表を関するプロセスを設けていた．このため，PL/0やPASCALのように宣言のネストを認める言語では記号表の検索を行う場合，最も内側の記号表管理プロセスから始めて，構文に沿って外側の記号表を順次検索する必要がある．一般にプログラムの特定のブロック中に現れる名前は限られており，しかも，同じ名前が何度も使用されることが多い．このことから，記号表管理プロセス間の通信量を減少させるために，記号表の管理を行なうプロセスに対して外側の記号表への検索結果をローカルな記号表管理プロセス内にキャッシュするように改良を行なった．

図6.5に記号表のキャッシングによる性能向上に関するグラフを示す．ここで，横軸はプロセッサ台数，縦軸は入力プログラムの1文字あたりの処理に要した平均クロック数を表している．ここでの入力プログラムには(1)と同様の約50行のテストプログラムを用いている．SMiSハードウェアに関しては，キャッシュメモリのヒット率を100ここで，記号表キャッシュ無しの版ではローカルに宣言しない名前の参照があった場合，より外側の記号表を管理するプロセスへ参照要求を送り受けとった結果を，参照側のプロセスへ送り返すという処理を毎回行なう．記号表キャッシュ有りの版では，ローカルに宣言しない名前の参照があった場合，最初はより外側の記号表を管理するプロセスへ参照要求を送り結果を受けとった後，この結果を局所的に管理する記号表に登録し，結果を参照側のプロセスへ送り返す．同一の名前に対して次にアクセスがあった場合は，前回記号表に登録された情報が参照され，参照側のプロセスに送り返される．これによって，より外側の記号表を管理するプロセスへのアクセスが緩和される．図からわかるように，プロセッサ1台の実行では記号表キャッシュ無しの場合1文字あたりの処理に約5000クロック要しているのに対し，記号表キャッシュ有りの場合には約4100クロックとなり，約18実行で比較すると，記号表キャッシュ有りの場合1文字あたりの処理に1300クロックを要しているにに対し，記号表キャッシュ有りの場合に1200クロックとなり約13下が見られるが，これも(1)の場合と同様に入力されるPL/0のソースプログラムが6.2.1で用いたものと比べて短いことによるものと考えられる．

6.3.3 仮想マシンプリミティブの評価

5章で述べた仮想マシンの各プリミティブの呼び出し回数と実行時間の割合を，プロセッサ5台の場合について表6.1に示す．ここでのテストプログラムは6.2.1の場合

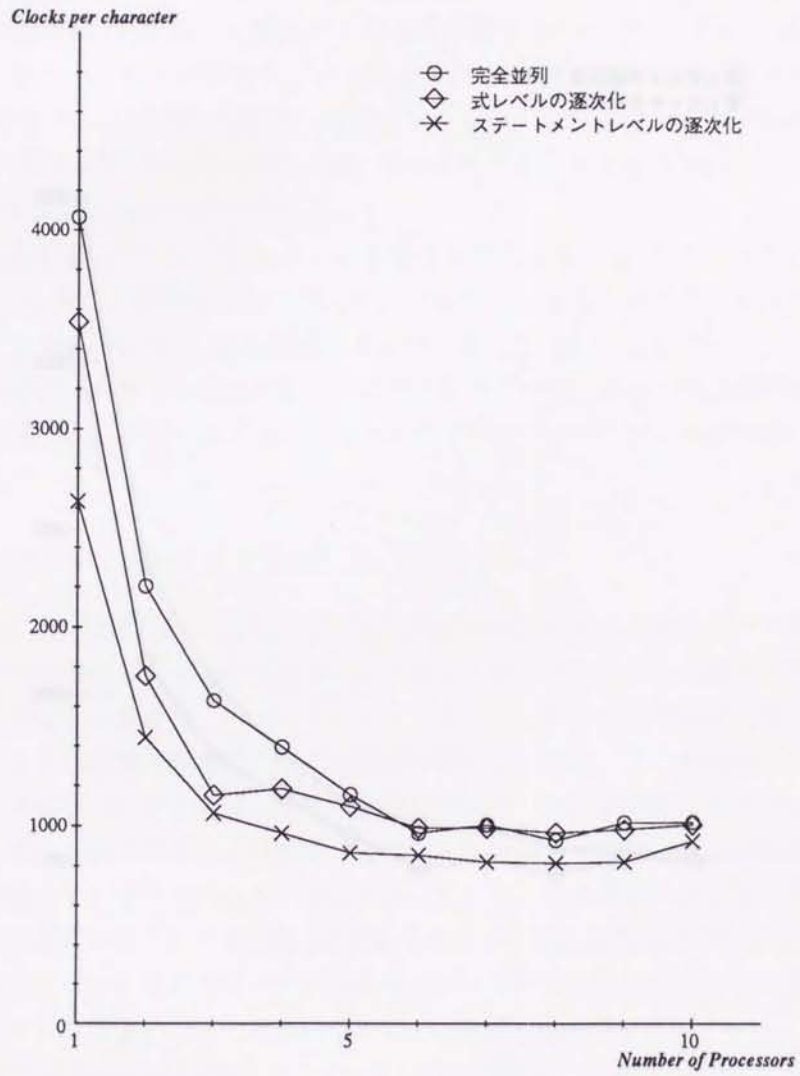

Clocks per character
4000
3000
2000
1000
0
1
5
10
Number of Processors
完全並列
式レベルの逐次化
ステートメントレベルの逐次化

図 6.4 逐次化の効果

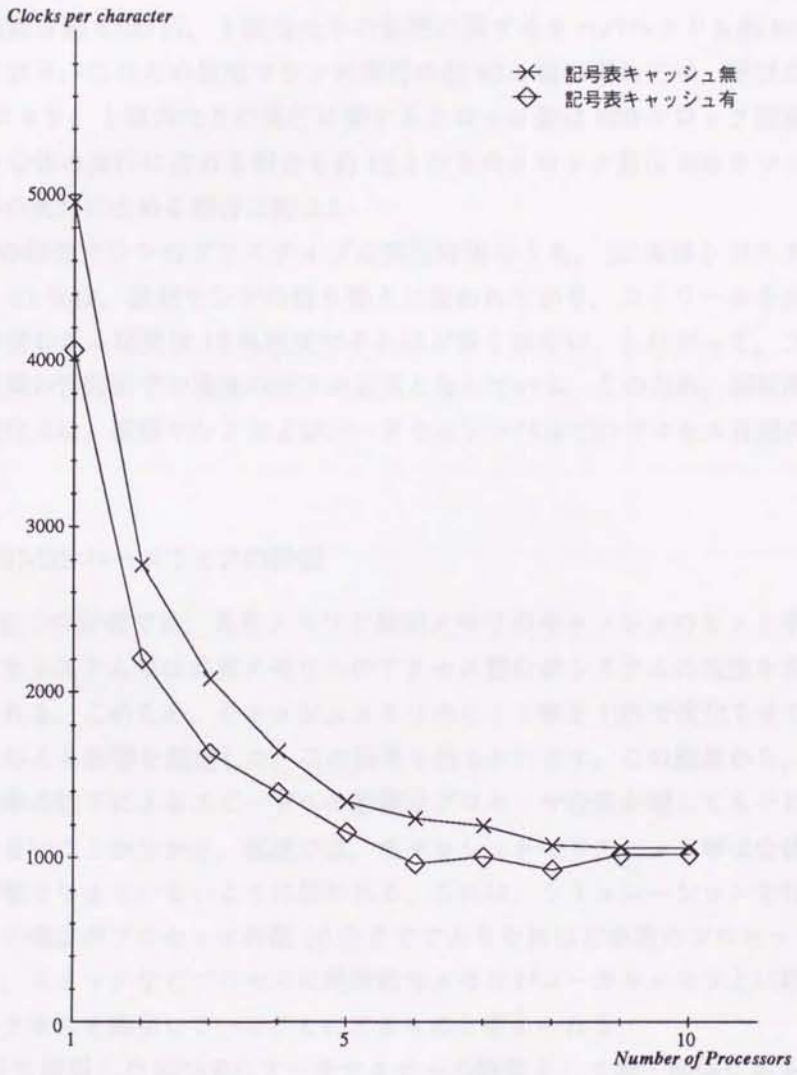


図 6.5 記号表キャッシュの効果

と同じく，約 250 行の 9 つの手続きを持つ PL/0 プログラムを用いている．また，SMiS ハードウェアに関しても，同様に，キャッシュメモリのヒット率を 100 を介した通信待ちが発生した場合，プロセス管理を行なうためのプリミティブがプロセス間通信を行なうプリミティブから呼び出されるが，この実行時間はプロセス管理のためのプリミティブに含まれている．現在の実現では仮想マシンプリミティブの 1 回あたりの平均的な実行時間は約 3,000 クロックとなっている．表に示すように，プロセス管理に関しては呼びだし回数は約 4700 回，1 回当たりの処理に要するオーバヘッドも約 8000 クロックとなっており，このため仮想マシンの実行の約 83.4 信に関しては，呼びだし回数は約 6600 クロック，1 回当たりの実行に要するクロック数は 800 クロック程度であり，仮想マシン全体の実行に占める割合も約 12.2 たりのクロック数は 800 クロック，仮想マシン全体の実行に占める割合は約 3.5

以上の仮想マシンのプリミティブの実行時間のうち，22 %はレジスタファイルの退避に，61 %は，仮想マシンの切り替えに使われており，ストリームを介した通信そのものに使われる時間は 12 %程度でそれほど多くはない．したがって，プロセス管理機構の実現が性能面での速度の低下の原因となっている．このため，細粒度の実行モデルの高速化には，仮想マシンおよびハードウェアレベルでのプロセス管理の高速化が重要である．

6.3.4 SMiS ハードウェアの評価

ここまでの評価では，共有メモリと局所メモリのキャッシュのヒット率をそれぞれ 100 セッサシステムでは共有メモリへのアクセス競合がシステムの性能を左右することが考えられる．このため，キャッシュメモリのヒット率を 100 で変化させて並列 PL/0 の実行に与える影響を測定した．この結果を図 6.8 に示す．この結果から，キャッシュのヒット率の低下によるスピードへの影響はプロセッサ台数が増してもそれほど大きくなっていないことが分かり，現状では，キャッシュメモリのヒット率は全体の性能にそれほど影響を与えていないように思われる．これは，シミュレーションを行なったハードウェアの構成がプロセッサ台数 10 台までであり，それほど多数のプロセッサではなかったことと，スタックなどプロセスに局所的なメモリがローカルメモリ上に取られ共有バスへのアクセスを緩和していることによるものと考えられる．

SMiS で採用した SPARC アーキテクチャの特徴として親子関係にある手続き間でレジスタのオーバーラップを可能とするレジスタ・ウィンドウを持つことが挙げられる．レジスタ・ウィンドウは高速な手続き呼び出しを可能にするが，一方ウィンドウ数の増加はコンテクスト切替え時に退避が必要となる情報を増やすことにもなる． 上で示した評価ではレジスタ・ウィンドウの段数をプログラムの開発に利用している SUN4 と同じく 7 としてシミュレーションを行なっていたが，これを 3 から 9 まで変化させてシミュレーションを行なった．この場合，レジスタ・ウィンドウの段数を小さくすると，仮想マシンを切替える際のオーバーヘッドが減少するが，手続き呼び出しのオーバーヘッドが増加する．反対に，レジスタ・ウィンドウの段数を大きくすると，仮想マシン切替のオーバーヘッドは大きくなるが，手続き呼び出しのオーバーヘッドが減少する．この

	呼び出し回数	1回当たりのクロック数	仮想マシンの実行に占める割合
プロセス管理	4,733	8,160	83.4%
メモリ管理	1,980	822	3.5%
プロセス間通信	6,612	851	12.2%
その他	2,310	170	0.9%

表 6.1 仮想マシンプリミティブの内訳

結果を図6.9に示す．ウィンドウ数が3から5の間はレジスタ・ウィンドウの効果が現れるが，それ以上ではそれほどの効果は得られていない．

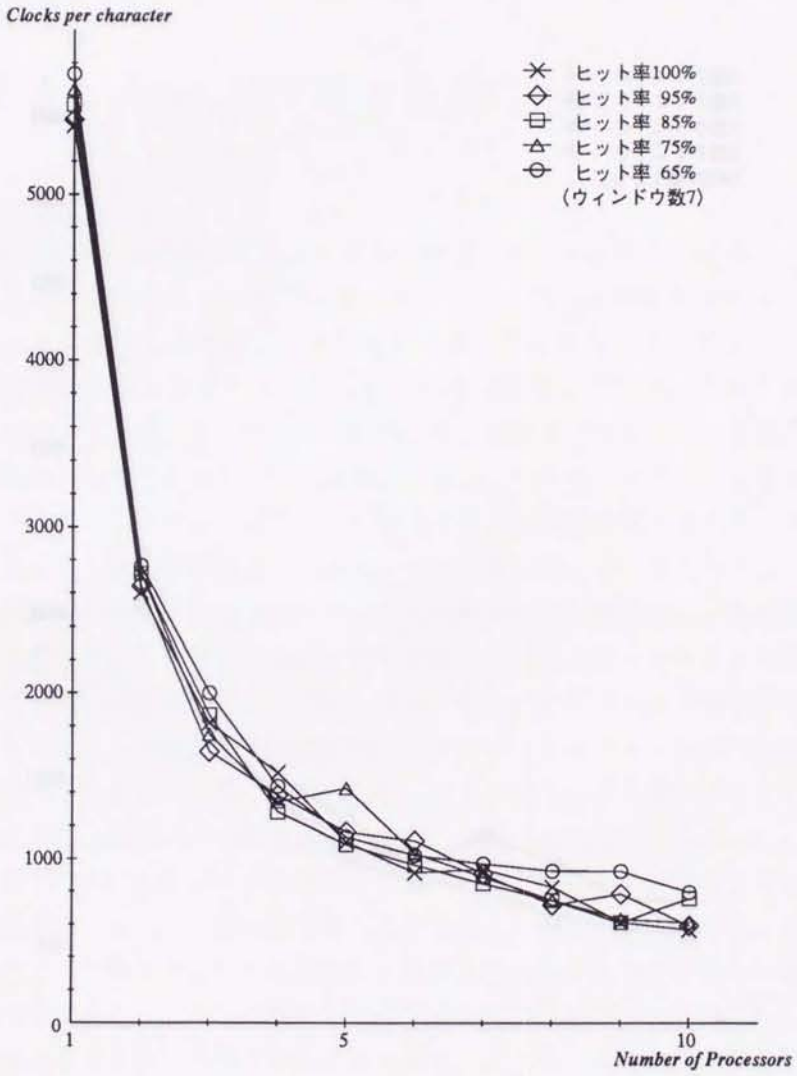


図 6.8 キャッシュメモリのヒット率が性能に与える影響

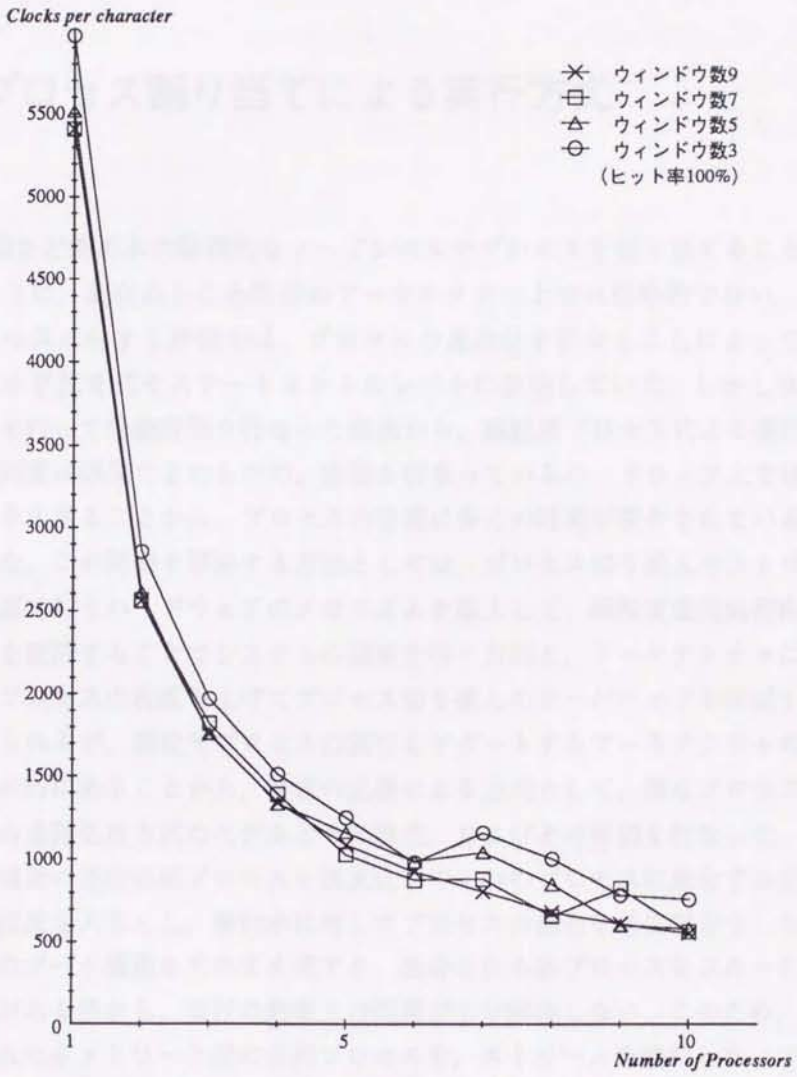


図 6.9 レジスタウィンドウ数が性能に与える影響

第 7 章

疎なプロセス割り当てによる実行方式

式の項など解析木の論理的なノードレベルでプロセスを割り当てることは，6章で示したように，現在のところ既存のアーキテクチャ上では効率的でない．このため細粒度プロセスに対する評価では，プロセスの逐次化を行なうことによって，プロセス生成の最小単位を式やステートメントのレベルに限定していた．しかしながら，具体的な実現を行って性能評価を行なった結果から，細粒度プロセスによる実行方式では，大きな並列度は確保できたものの，実験を行なっているハードウェア上ではプロセスの粒度が小さすぎることから，プロセスの管理に多くの時間が費やされているという結果が得られた．この問題を解決する方法としては，プロセス切り換えやストリームのアクセスを高速に行うハードウェアのメカニズムを導入して，細粒度並列処理向きのアーキテクチャを設計することでシステムの開発を行う方向と，アーキテクチャに変更を加えないで，プロセスの粒度を上げてプロセス切り換えのオーバヘッドを軽減する方向の2つが考えられるが，細粒度プロセスの実行をサポートするアーキテクチャの研究は本研究の視野の外にあることから，後者の立場による方式として，疎なプロセス割り当てによる意味の並列処理方式のモデルとその実現，およびその評価を行なった．ここでの実現では，複数の意味処理プロセスを逐次化して1つのプロセスに統合する方式によりプロセスの粒度を大きくし，解析木に対してプロセスの割当を疎に行なう．ただし，ストリーム型のデータ構造をそのまま残すと，統合される各プロセスをコルーチン的に実現する必要がある事から，実行の効率上の問題が十分解決しない．このため，ストリームで結合されたネットワーク型の並列プロセスを，ストリームを使用しないで逐次的に処理できるアルゴリズムに変換する．この変換は，文法に対する意味解析プロセスの定義から，逐次的に処理可能な部分解析木を分割し，この部分解析木を独立したプロセスによって実行することにより行なう．

7.1 実行方式の概要

疎なプロセス割り当てによる実行モデルでは，プロセスとストリームによる並列意味処理の基本的な枠組みは，これまでの細粒度並列処理モデルで用いている枠組みをそのまま使用し，オーバヘッドの少ない方式で意味処理プロセスを統合しプロセスの粒度

を上げるためのアルゴリズムを新たに導入する．

7.1.1 並列プロセスの逐次化の条件

細粒度プロセスによる実行方式におけるように，並列な意味処理プロセスを逐次化して１つのプロセスに統合しても，統合に要する実行時のオーバヘッドが大きいと，より大きな単位でプロセスの統合処理を行なった場合に，実行の効率上の問題が十分解決しない．そこで，ストリームで結合されたネットワーク型の複数のプロセスを，ストリームを使用しないで逐次的に処理できるアルゴリズムに変換することを考える．この変換は，任意のネットワークについて適用できるわけではないので，容易に変換が可能なサブネットワークを抽出しなくてはならない．また，この抽出にかかる実行時のコストが大きすぎることは望ましくないない．ここでは，具体的には，プロセスを結合しているストリームが次のような条件を満足する場合をこの変換の対象として考える．まず，ここでは解析木等の中間的なデータ構造の保持は仮定せず，プロセスの生成処理は，いままでと同様に構文解析による解析木の認識と同時に行われるものとする．以上のような条件のもとで，解析木上での意味解析処理を行なうプロセスのストリームへの入出力関係を，属性文法における属性の依存関係と同様にとらえ，統合対象となるプロセスからの入力に関しては，解析木の親あるいは，解析木の左側の兄弟に属するプロセスからの出力のみを許すこととする．また，統合対象となるプロセス間でストリームの依存関係にループがないものとする．以上の関係をみたす部分解析木にたいする実行は，統合されたプロセス間のストリームを完全に評価された形で受け渡すことによって，単一のプロセスによる逐次的な評価に変換可能であり，これに従ってプロセスの統合を行う．統合されるプロセス以外のプロセスからの，ストリームによる入力および出力に関しては，上で述べたような制限は適用されない．

この方式を用いることによって，細粒度モデルにおいて解析木のノードの認識に対応して生成されていたプロセス群のうち，部分解析木の深さ優先の左から右への走査，つまり逐次的な構文解析動作列の入力によって，ストリームを全要素が評価された値の列として受け渡しを行うことが可能な領域に分割し，これら細粒度モデルにおけるプロセス群を，実行時のオーバヘッドの少ない方式で部分解析木上の単一のプロセスに変換することができる．ここでの条件を満たさないプロセスに関しては，細粒度の実行モデルと同様に，独立したプロセスとして扱われる．

統合されるプロセスの割り当ては，上で示した制限に従うプロセスに対応した部分解析木のうち，ストリームの入出力関係に互いに依存関係のないプロセスからなる，独立に実行可能な部分木にさらに分割することが可能であるが，ここではプロセスの粒度をできるだけ大きくし，実行時のオーバヘッドを低減するという観点から，部分解析木の再分割は行なわないものとする．

ここで示すようなストリームを使用しない逐次化処理は，もともとストリームで結合されていたプロセスの集合ネットワークが，言語処理系で抽象構文木の様な中間構造を明示的に作成する処理方式に対応しているとみなすことができるのに対し，解析木を明示的に作成しない１パス型の処理方式に対応していると考えることもできる．

7.1.2 構文によるプロセスの統合制御

7.1節で述べた逐次化の対象となる部分解析木の選択は，部分木解析器に対応した非終端記号列の前後にプロセスの統合の開始と終了を表すための特殊なマーカ用の非終端記号を導入することにより，構文解析処理に同期させて行なうことが可能である．例えば，

X → A B C D E

のような文法に対して，A, B に対応した部分解析木と D，E に対応した部分解析木に属したプロセス群をそれぞれ1プロセスにまとめるには，上の文法を次のように変形する．

X → SB A B SE C SB D E SE
SB → ϵ
SE → ϵ

ここで，SB と SE はそれぞれプロセスの統合の開始と終了に対応した非終端記号である．意味解プロセスの生成処理を行なう制御プロセスは，プロセスの統合の開始に対応したマーカの還元動作を構文解析器から受け取ると，新たにプロセスを生成して，対応した終了マーカの還元動作までの構文解析動作を読み飛ばし，その構文解析動作を新たに生成したプロセスに受け渡す．新たに生成されたプロセスでは受けとった構文解析動作を解釈しながら逐次的に意味解析処理を行なう．

このプロセス統合の為のマーカの挿入点は，文法に対するプロセスの入出力関係の定義を与えることにより7.1節で示した統合のための条件から得る事ができる．このように，文法によってプロセスの統合を制御することにより，実行時のプロセス統合に要するオーバヘッドを低減することがかのうとなる．

ここで導入した手法の問題点としては，与えられた文法に対してプロセスの統合を制御するための非終端記号を導入することにより，変形後の文法のクラスが構文解析部で扱うことの可能なクラスに収まらない可能性が生じることと，文法および意味処理プロセスの記述によっては，並列実行が可能な部分解析器が得られない場合があることが挙げられる．しかし，PL/0 コンパイラの記述等における経験では，意味処理の記述に注意を払えば，これらは特に問題とはならないと考えている．

7.1.3 プロセスの生成処理

本節で示す疎なプロセス割り当てによる実行モデルにおいても，プロセスの生成処理の実現は基本的に6.2節で述べた基本的に実現と同様である．すなわち，図7.1に示すように構文解析の結果として生成される構文解析情報は，制御プロセスと呼ばれるプロセスの生成処理を司るプロセスによって受理され，逐次化の指定が行なわれていない通常のプロセスに関しては，細粒度の実行モデルと同様にプロセスの生成処理とストリームによる結合処理を行なう．ただし，制御プロセスは，逐次化の開始に対応する構文解析動作を受理すると，新たにプロセスを生成し，対応する逐次化の終了までの構

文解析動作を読み飛ばし，これを先に生成したプロセスに引き渡す．新たに生成されたプロセスでは，受けとった構文解析動作を順に解釈し，これに対応して定義された仮想的なプロセスに対する意味操作を実現する関数を実行することにより，部分的な解析木のボトムアップな構文の認識に対応して逐次的な意味処理の実行を行なう．これら仮想的なプロセス間での情報の受渡しは，部分的な解析木の認識に対応した構文解析状態を保存するためのスタックと同期して操作される属性評価用のスタックを介して行なわれる．なお，図 7.1 では白抜きの丸でプロセスを，実線の矢印によって制御プロセスによるプロセスの生成過程を，破線によって新たに生成されたプロセスへの部分解析木に対応する構文解析動作の受渡しを表している．

7.2 評価

細粒度方式の処理と疎なプロセス割り当てによる方式の処理のそれぞれについて，5 章で述べた SMiS シミュレータ上でシミュレーションを行った．なお，本章での細粒度の実行モデルに対するシミュレーションには速度面で改良を加えた仮想マシンを使用し，さらにプロセスの逐次化によるプロセス数の減少や記号表キャッシュなどを実現に採り入れているため，6 章で示した実現と比較して高速化されている．

疎なプロセス割り当てによる実行モデルの具体的な実現は，6 章で示した PL/0 コンパイラを改良することにより行っている．この実現で，並列プロセスの統合を行なうための条件を満足する部分は，PL/0 コンパイラの細粒度プロセスによる実行方式の記述中では，ブロックのステートメント部，宣言部の定数宣言と変数宣言部に対応する部分の計 3ヶ所のみであった．したがって，疎なプロセス割り当てによる実現では，以上の 3ヶ所に対応してプロセスの統合の開始と終了を制御するマーカを挿入している．

プロセッサ 5 台で実行した場合における，プロセス数，プロセス切り替えの間の平均的な実行クロック数であるランレングス，プロセッサ毎のプロセス切り替えの平均数を表 7.1 に示す．なお，ここでの評価では，テスト・プログラムとして約 250 行の PL/0 プログラムを使用している．ここで用いたテストプログラムは 9 つの手続きを持つプログラムであり，SMiS ハードウェアに関しては，キャッシュメモリのヒット率を 100%，レジスタ・ウィンドウの段数を 8 としている．

表からわかるように，細粒度実行モデルの処理に対して，疎なプロセス割り当てによる処理ではプロセス数が，168 から 44 となり 26% に減少している．プロセス切り替え間の平均的な実行クロック数に対応するランレングスは細粒度の実行方式が約 3,800 であるのに対して，疎なプロセス割り当てによる方式では約 6,100 となり，1.6 倍となっている．また，プロセス切り替えの平均数は 382 から 55 となり，14% に減少している．プロセス数の減少に比べてプロセス切り替えの間の平均的なクロック数に対応するランレングスの増加が少ないのは仮想マシンの実行によるプロセス間の待ち合わせと，統合されたプロセスでの構文解析動作の解釈と属性評価スタックなどのデータ構造の操作が影響している．また，ここでの全体の処理時間は，約 112 万クロックから約 47 万クロックとなり，ほぼ 2.4 倍の速度向上となっている．

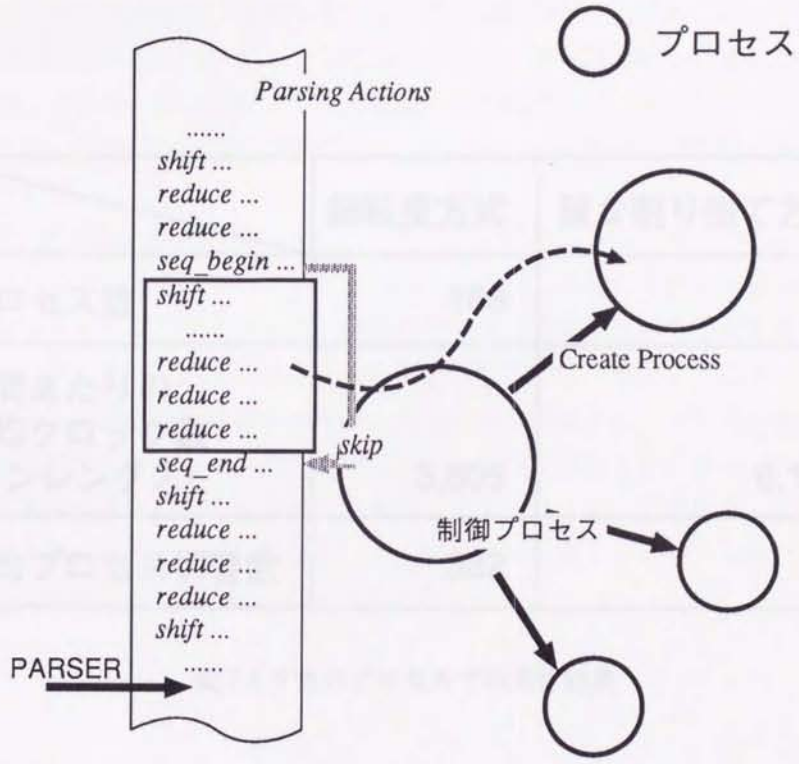


図 7.1 疎なプロセス割り当てによるプロセスの生成

	細粒度方式	疎な割り当て方式
プロセス数	168	44
区間あたりの 平均クロック数 (ランレングス)	3,805	6,101
平均プロセス切替数	382	55

表 7.1 5台のプロセスでの実行結果

次に，上と同様なサンプルプログラムを，プロセッサ台数を1から10まで変えて実行した場合の実行結果をそれぞれの処理方式について図7.2に示す．図7.2で，横軸はプロセッサ台数を表し，縦軸は入力プログラムの1文字あたりの意味処理に要したクロック数を示している．各データは，それぞれ仮想マシンと意味処理のクロック数に対応している．図7.2から分かるように，意味処理の実行クロック数はプロセッサ台数に比例して減少しているが，プロセッサ数の増加により仮想マシンの実行に要している時間が増加し，実質的な速度向上が低く抑えられている．以上の結果から，構文レベルのプロセスの統合制御に基づいた疎なプロセス割り当て方式を導入することにより，1.9倍から4.5倍の速度向上を得ることができた．

ここで，細粒度実行モデルに比較して，疎なプロセス割り当てによる実現での意味処理に要している処理時間が増加しているのは，細粒度実行モデルにおいて複数の独立したプロセスによって仮想マシン上に実現されていた意味処理が1つのプロセスに統合された結果，プロセス間通信やプロセス生成などの処理が統合されたプロセスの意味処理に吸収され，構文解析動作の解釈や属性評価スタックの操作に置き換えられたことによる．

疎なプロセス割り当て処理によって，プロセス管理のためのオーバーヘッドを減少させたことにより，疎なプロセス割り当てによる処理では実行性能が大幅に改善されたが，並列度は減少する結果となった．構造エディタの使用を前提とした解析木が内部構造として保持されている環境では，既にある解析木に対して最適なプロセスの割当を事前に計算することが可能な場合もあるが，バッチ的な環境では，プロセス割当の最適化処理を実行時に行なうことは負荷の増加を伴うという面で好ましくない．このような点から，本方式は，我々が想定しているようなパイプライン型の構成をとった並列コンパイラ・システム上の意味処理の手法として，細粒度の実行をサポートしないハードウェアに対する実行方式として有効であると考える．

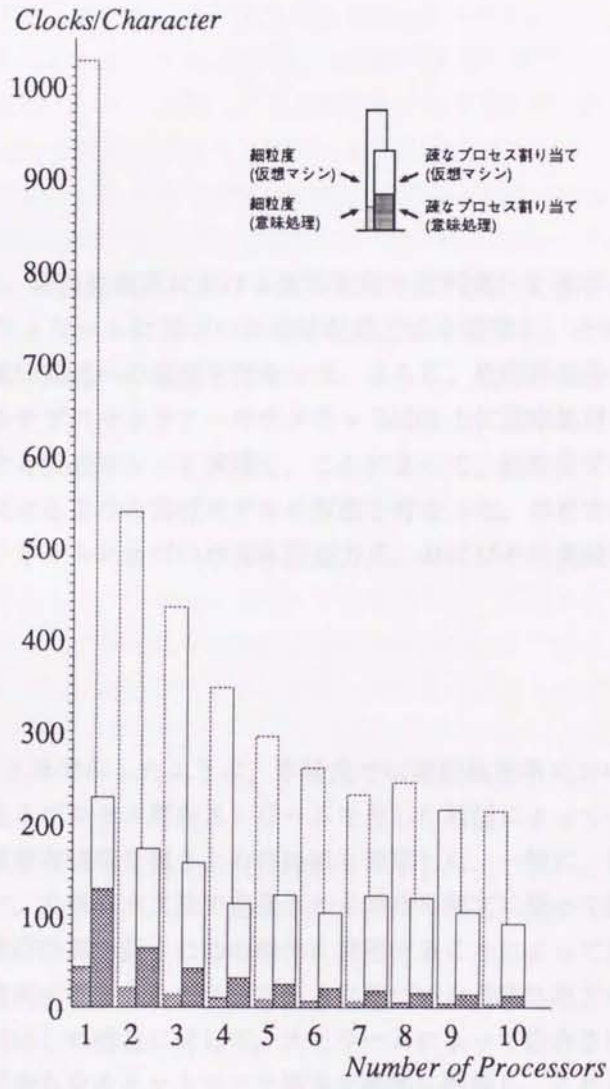


図 7.2 各方式の実行結果

第 8 章

考察

本論文では，言語処理系における意味処理を並列実行を基本として構成するための方式として，ストリームに基づいた意味記述方式を提案し，その記述言語の実現および，具体的な意味記述への適用を行なった．さらに，性能評価のための実行評価システムとして，マルチプロセッサアーキテクチャ SMiS 上に意味処理を行なうプロセスの実行環境を構成する仮想マシンを実現し，これによって，細粒度プロセスおよび疎なプロセス割り当てによる 2 つの実行モデルの評価を行なった．本章では，以上のような本論文で示したストリームに基づいた意味記述方式，およびその実現等に関して考察を行なう．

8.1 記述

2 章および 3 章で示したように，本研究では言語処理系における意味処理を自律的な実行主体であるプロセス群のストリームを介した通信によってモデル化することにより，言語の文脈依存情報を扱うための枠組を提案した．一般に，言語に対する文脈依存情報を扱うには，文脈自由文法で定義される言語の構文に従って認識された対象構造の従って，文脈依存情報を扱うための操作を適用することによって意味処理を行なう構文主導型の翻訳方式が採られる．ストリームに基づいた意味処理方式では，入力を構文解析した結果に対応した構造に対して，ストリームによって結合された自律的実行主体であるプロセス群からなるネットワーク構造を動的に構築し，これら自律的実行主体であるプロセスのストリームを介した通信によって，認識された構造の文脈依存情報に対する意味処理の適用を行なう．

従来の言語処理系に対する意味記述のための枠組としては，一般に，文法によって指定された入力構造の認識に対応して意味処理を行なう手続きを実行する動作ルーチンによる方式が用いられることが多い．このような動作ルーチン方式による記述は，基本的に入力構造のトップダウンあるいはボトムアップな認識が行なわれるのと同時に意味手続きの呼びだしが行なわれるため，入力とそれに対して適用される手続きの関係が明瞭であることから，手書きのコンパイラや YACC のような生成系によって採用されて

いる．このような動作ルーチンに基づいた方式では，一般に入力の構造に従って指定された手続きを適用することにより，大域的な情報を書き換えることによって文脈依存情報の収集と意味処理を実行する手続き間での情報の受け渡しが行なわれる．このため，動作ルーチンに基づいた方式では入力のある時点で実行される手続きでは，その後に現れる入力構造から得られる文脈情報を利用することができない．例えば，多くのプログラミング言語において，ラベルの前方参照を解決するためにはラベルの参照部において定義部の情報を得るには，入力に対応した構造の後方から前方への定義情報の転送が必要となるが，動作ルーチンに基づいた方式ではこれを直接的に扱うことは難しく，バックパッチなど特殊な技法の利用や，コンパイラの構成をマルチパス型にすることによって問題を回避することなどが，言語処理系の記述時に要求されることとなる．また，動作ルーチンに基づいた方式では，手続き間の文脈依存情報の受け渡しは基本的に大域的な情報の書き換えによって行なわれるが，これは言語処理系記述の複雑さを増し，モジュール性を損ねる原因となる．このため，言語処理系の開発や保守を難しくする原因となる．同様に，これは意味処理を並列に行なうことを考えた場合の障害ともなる．

動作ルーチンによる記述方式に対して，言語処理系の形式的な意味記述方式として属性文法に基づいた方式が知られている．この方式では，構文によって定義される構造に対して，解析木上のノードに属性と呼ばれる値の集合を考え，その値を定義する関数の定義関係を与えることによって意味の記述を行なう．属性文法による方式は，その評価や定義のモデルによって様々な階層に分類されるが，その多くは属性の再帰的な定義を許さないことから，動的意味を含んだ多くの対象の記述への適用が難しい．同様の理由により入出力や書き換えを行なうような記号表の操作などの副作用を伴った処理の記述も，その基本的な枠組からは扱うことが難しくなっている．また，文脈依存情報に対応する全てのデータは，解析木上のノードの属性の値の入力と出力としてモデル化され，これによって各生成規則に対して定義された意味規則の独立性は保たれているが，反面でこれは意味規則によって扱われる全てのデータ構造が外部に開かれていることを意味しており，これが意味記述のモジュール性を一面で損なう結果となっている．また，文脈依存情報の受渡しは解析木の構造に強く縛られているため解析木の構造を越えた情報の受渡しは面倒である．属性文法に基づいたシステムにおいて並列処理を行なうことを考えた場合，独立して定義される属性を並列に計算することによって並列な実行を行なうことが可能である．動作ルーチンに基づいた方式とは異なり，属性文法に基づいた方式では文法の生成規則に対して定義された属性の値の決定とその解析木上のノード間のによって意味処理が行なわれるため，大域的な情報の変更の競合による問題は発生しない．ただし，属性文法では属性の値を適当な順序に従って決定することによって実行が行なわれることから，意味処理において文脈依存情報として扱われるデータに内在する並列性の取り扱いが制限されることとなる．このため，例えば，4章で示したパイプライン型のコード生成に見られるような並列性を属性文法に基づいたシステムで扱うことは難しい．

ストリームに基づいた意味記述方式では，入力に対応した構造に対して，その文脈依存情報に対応して自律的な実行主体を割り当て，これらをストリームと呼ばれる通信路で結合し，これによりストリームによって結合された動的な実行主体のネットワークを動的に構成することによって意味処理を実行する．ここで，ストリームに基づいた意味記述方式においても，動作ルーチンに基づいた方式のように，解析木の構築という手順を踏まず，入力の解析を行なうと同時に意味操作の実行を行なうことができるが，動作ルーチンに基づいた方式のような記述上の制限は必要とされない．これは，ストリームに基づいた意味記述方式では未解析な構造に対応した文脈依存情報は，未解析な構造に対応して生成されるプロセスに結合されるストリームとして表現されることから，入力の後方の構造から得られる情報の参照の問題は，ストリームからの入力による同期へと変換され，言語処理系記述を行なう際にバックパッチやマルチパス型の構成は不要となる．また，ストリームに基づいた意味記述方式では，意味処理を自律的な実行主体であるプロセスと，そのストリームを介した通信によって記述することから，入出力など必要な副作用はプロセス中に吸収される．また，ストリームに基づいた意味記述方式では，文法に附属した意味規則においてプロセスの結合関係のみを与えることから，記述のモジュール性が確保される．このため，具象構文の変更や，同様な言語コンストラクトを持った複数の言語への適用に柔軟に対処することが可能となる．

8.2 応用

本論文で示したストリームに基づいた意味記述方式では，このような意味処理を，自律的な実行主体の通信として定義することにより，種々の意味記述に対してそのモジュール化された記述と，意味処理の並列実行を行なう際の並列性の抽出を可能としている．プログラミング言語のような，構造を持ったデータに対する文脈依存の情報の扱いは，基本的に静的意味と動的意味に類別される．静的意味は入力の静的な構造から得られる情報を他の情報へ変換する枠組であるのに対して，動的な意味では，入力によって指定された構造に対して，それによって指定された動的な処理を実行するための枠組である．本節では，ストリームに基づいた意味記述方式の応用を，静的意味と動的意味それぞれに対して考える．

(1) 静的意味

静的意味は入力から出力への変換としてモデル化することが可能であり，コンパイラやフィルタ型のツールをその対象としている．このため，本論文で示したようなストリームに基づいた意味記述方式の静的意味記述に対する適用は，コンパイラのような言語処理系だけでなく，構造を持ったデータに対するフィルタ型ツールの並列処理による高速化の可能性を示している．例えば，3章のSSGLによる記述例で示した入力された数値列のソーティングを行なう文法と意味の記述のように，構造を持ったデータに対する操作の並列な適用に本論文で示したような方式を利用することも可能である．コンパイラを含んだ，このようなフィルタ型のツールはプログラム開発の局面で多用される

ものであり，基本的にユーザが実行に介入しないバッチ型の利用形態をとる．これらのツールの実行時間はユーザに対しては純粋な待ち時間であり，その高速化はプログラム開発時のプログラマの待ち時間の減少に貢献する．特に，近年のグラフィカルなユーザインタフェイスの広範な普及などによりプログラムは大規模化を続けており，近い将来広く利用可能となると予想されるマルチプロセッサシステム上でのこれらのツールの高速化はプログラマのプログラミングへの集中を阻害しない快適なプログラミング環境を構成する礎石となると考える．

また，前節でも述べたようにストリームに基づいた意味記述方式では，1パス型の構成で，名前の前方参照をバックパッチなど特殊な技法やマルチパス型の構成を用いることなく扱うことが可能になるなど，言語記述を行なう上で自然な記述を行なうことができる．さらに，ストリームに基づいた意味記述方式では，BNF などによって与えられる言語の具象構文と，具体的な意味処理を行なう自律的な実行主体であるプロセスが明確に分離されている．このため，ALGOL 型のスコープ規則を扱う記号表管理プロセスや，while 文や代入文など，多くのプログラミング言語で共通な言語コンストラクトに対するプロセスの記述をライブラリ化することにより，言語処理系のプロトタイピングや，言語記述のための部品の共用化を行なうことが可能となる．

(2) 動的意味

動的意味は，プログラム等の実行時の動的な情報を記述するための枠組である．動的意味記述を行う枠組みを意味記述モデルにおいて提供することにより，インタープリタや言語指向環境，ユーザインタフェイスの記述などユーザとの対話を伴った対象への応用が可能となる．このような動的意味を記述するための枠組としては様々な方式が研究されている [Ambriora 85][Sato 86][Kaiser 89][Watt 86] が，ストリームに基づいた意味記述方式では，言語処理系を自律的な実行主体であるプロセス群によって構成することにより，言語処理系に対する動的意味のさまざまな記述を自然な形で与えることが可能となる．

プログラミング言語などの動的意味は，基本的に言語コンストラクトの実行による実行環境の変化としてモデル化される．ストリームに基づいた意味記述方式においては，各言語コンストラクトに対応して定義されたプロセス間を定義される実行順序に従ってストリームで結合し，実行時の環境をこれらのプロセス間でストリームを介して受け渡すことにより動的意味の記述を与えることができる．また，入出力などの言語プリミティブに対応した副作用の扱いは対応したプロセス内に封じ込めることができる．インタプリタなどに対する動的意味の記述においては，デバッガなどからユーザが意味の実行を監視および制御することが重要であるが，ストリームに基づいた意味処理方式においては，制御対象の言語コンストラクトに対応したプロセスを制御するストリーム上の実行時環境の変化を監視することによりプログラムの動的な実行を監視することができる．同様に，プロセス間で受け渡される実行時環境をユーザからの指示により書き換えるようなプロセスを，デバッグ対象となるプログラムコンストラクトに対応したプ

ロセスを制御するストリームを切断するような形で挟み込むことによりデバッガなどの構成も可能となる．

多くの手続き型言語では実行順序や，言語プリミティブによる実行状態変化の伝搬などが言語の定義として与えられている．このため，静的意味の場合と異なり，動的意味記述では言語の定義内でその並列な実行を行なうことは難しい．このような理由から，静的意味の場合のような言語処理系の高速化という利点は，ストリームに基づいた意味記述による動的意味の記述では，一般に得られない．ただし，最近では多くの並列プログラミング言語が開発されると共に，逐次型の言語に対してもコルーチンや例外処理など並列な解釈が有効な言語や言語コンストラクトがプログラミング言語に採り入れられるようになっている．ストリームに基づいた意味処理方式では並列実行を基本とした実行もモデルを採用しているためこのような並列な実行対象の記述にも親和が高く，自然な記述を与えることができる．また，静的意味記述の場合と同様に，動的意味記述においても，多くの言語に共通な言語コンストラクトに対するプロセスの記述を部品として用意することにより，言語処理系のプロトタイピングや，部品の共有化を行なうことが可能となる．

8.3 実現

ストリームに基づいた意味記述の具体的な記述のための言語としては，3章で述べたようにSSGLと呼ばれる言語を実現した．SSGLはBNFによる文法と解析木上のプロセスの結合による意味規則を与えることによって意味記述を行なうための言語である．前節でも述べたように，SSGLではBNFによる言語の具象構文の定義と，プロセスの定義が分離されており，4章で示したようなさまざまな言語記述を行なう上で有用であった．また，並列コンパイラの記述など開発の初期では意味処理を実行するプロセスの記述はC言語で直接与えていた．後に，プロセス記述のために設計した言語SSDLは，C言語による意味解析プロセスの記述の経験をもとに，動的なプロセスの生成やストリームによる通信を高レベルで与える機構を取り入れて設計と実現を行った．SSDLは主にインタプリタ記述を対象にした動的意味記述の実験に利用されたが，SSGL記述から利用されるプロセスの記述を高水準言語によって与えることにより，記述の簡略化や効率的な開発に有効であった．

本論文における，ストリームに基づいた意味処理の実現は，基本的に意味処理プロセスに対して軽量プロセスを割り当て，ストリームに対する値の送受信による変化の伝搬によって，プロセスの切り替えを行う方式に基づいている．ここでの多重プロセスの実現は，コルーチンに基づいて軽量プロセスを切り替える方式をマルチプロセッサシステムに拡張したものに基づいている．これは，本論文の6章および5章の実行モデルとその評価が，コンパイラの意味処理のマルチプロセッサシステム上での実行による高速化をその対象としていたことによる．

5章でも述べたように，この具体的な実現は特殊なハードウェアを持たないハードウェア上での実現を想定し，現状のハードウェアアーキテクチャを利用した一般的な構

成の密結合型マルチプロセッサアーキテクチャ上に，ハードウェアに依存しない仮想マシンを構築することによって行われている．仮想マシンは5章で示したようなプロセスの動的な生成，ストリームによるプロセス間通信，共有メモリの割当などを意味処理を行うプロセスに提供する．このように，意味処理の実行を仮想的なハードウェアと意味処理プロセスとに分割することは，プロセスの実現や通信機構の詳細を実行方式の実現から隔離し，プロセスの切り替えや通信頻度など意味処理実行の動的な様子のシミュレーションによる把握や，これに基づいた仮想マシンの改良や実行モデルの改良に有効であった．

細粒度プロセスによる実行モデルは，ストリームに基づいた意味記述方式の自然な実現であり，基本的にSSGLで定義された生成規則とそれに付随したプロセスの結合規則に従って，入力の認識に従って意味解析処理を行うプロセスの生成を行い，ストリームを介した通信を行うことによって実行を行う．このため，6章で示したように細粒度プロセスによるでは，複数のプロセスへの分割によって高い並列度が得られている．しかしながら，現状の実現では，仮想マシン部の実行のオーバヘッドによって，性能面での向上はYACCによる動作ルーチンに基づいた逐次的な実現と比較すると不十分なものとなっている．ただし，8.1節などでも述べたようにストリームに基づいた意味記述では動作ルーチンに基づいた方式と比較して記述の明快さや記述力の点で優位である点と，複数のプロセスの実行に伴った仮想マシンのオーバヘッドに関してはハードウェアアーキテクチャの進歩によって減少する可能性が高いことをここでは指摘しておきたい．

疎なプロセス割り当てによる実行モデルは，上で示したような，細粒度の実行モデルにおける性能面での問題点を解消することを目的として開発した実行モデルである．ここでは，細粒度プロセスによる実行モデルで複数のプロセスとして独立に実行されていたプロセスを1つのプロセスとして実行することによって並列性を犠牲にすることにより，複数のプロセスの実行を逐次化し仮想マシンのオーバヘッドを減少させている．これにより，プロセッサ数の増加に対する速度の向上は低く抑えられているが，プロセスの統合によって性能は向上している．この疎なプロセス割り当による実行モデルでは，プロセスの動的な統合処理に要するオーバヘッドを減少させるため，統合の制御を構文レベルで行っている．

ここでの2つの実行モデルは，対立するモデルではなく補完的なものであると考える．すなわち，疎なプロセス割り当による実行モデルは，本論文で意味処理実行のためのハードウェア環境として使用した，細粒度プロセスの実行を支援しないマルチプロセッサ環境での実行方式に向き，細粒度プロセスによる実行モデルは細粒度プロセスの効率的な実行をサポートするようなハードウェア環境での実行に向くと考える．

8.4 今後の課題

文法および意味規則の定義言語であるSSGLと意味処理を行うプロセスを記述するための言語であるSSDLに関してはその実現や，さまざまな対象への適用など多くの

課題が残されている．本論文の６章および７章で示した，細粒度の実行モデルと，これを発展させた疎なプロセス割り当てによる実行モデルは，マルチプロセッサアーキテクチャ上での効率的な並列実行を目的としてしている．これらの実行モデルは，基本的に静的意味記述によるコンパイラなどのフィルタ型のツールを対象とするものであり，単一プロセッサ上での動的意味などの実行モデルとしては適当とは言えない．このため，動的意味などの記述とその実現に関しては，解析木や，意味処理を行なうプロセスのネットワークに対応した構造を構成して，これらを動的に辿ることにより実行を行うモデルなど対象に即した実行モデルを考える必要がある．また，疎なプロセス割当による実行モデルでは，プロセスの動的な統合を構文によって制御しているが，これは入力の構造によっては過度の逐次化を招く可能性も持っている．このため，文脈情報によるプロセス生成の制御などより細かなプロセス生成制御を行う手法を確立する必要がある．

本論文で対象とした研究では，基本的にハードウェアアーキテクチャを研究対象としていない．近年は細粒度の並列処理方式をサポートするハードウェアやソフトウェアの研究が精力的に行なわれており，このような研究の結果を援用することにより本論文で示したような言語処理系の並列な動作をより効果的におこなうことができる可能性がある．

第 9 章

おわりに

言語処理系における技法はプログラミング言語に対するバッチ指向の環境への適用を中心に発展してきた．本研究は，このような方向に対して，言語処理系の自然な記述と，並列実行のための枠組を研究することを目的として研究を進めてきた．この目的に沿って，本論文では言語処理系における意味処理の並列な記述と実行のための枠組として，ストリームににによって結合されたプロセスの相互作用として記述するモデルを提案した．このモデルに従って，意味記述およびプロセス記述用の言語を作成すると同時に，ストリームに基づいた意味記述の具体例として，実際に言語処理系における様々な意味の記述を行なった．さらに，意味処理の並列実行の詳細な計測を行なうために，意味処理実行用の環境を仮想マシンとして定義し実現した．この仮想マシン上でストリームに基づいた意味処理モデルの自然な実現である細粒度プロセスによるモデルを実現しその評価を行った．続いて，ここで顕在化したプロセス切替等による性能の低下に対処するため，解析木に対してプロセスの割り当てを疎に行なうモデルの実現と評価を行なった．

ユーザインタフェイス等の急速な普及によりプログラムは年々大規模化する傾向にある．また，並列システムや分散システム等に対する最適化や，高水準言語と様々なハードウェアに対する種々の最適化技法はコストの高い処理である．また，プログラミング・パラダイムの進展によりプログラミング言語はより高水準な記述を支援するようになっている．ハードウェアシステムの速度が向上を続けているとはいえ，言語処理系に対する高速化の要求は留まる所がない．すなわち，ハードウェアの速度の向上を上回る形で言語処理系に対してより複雑な処理が要求されるという循環が繰り返されている．

従来の言語処理系記述方式は，基本的に言語処理系記述の簡潔さや記述力を犠牲にすることによって高速化を行なってきた．例えば，バックパッチなどの技法はこの延長上に存在する．しかしながら，これは言語処理系記述者に負担をかけることであり，望ましいことではない．このことから，ストリームに基づいた意味記述ではできるだけ汎用的な目的に適用可能なモデルを設定することにより，広範囲の目的に適用可能なシス

テムを目指している．例えば，言語処理系記述のための形式的な枠組として知られる属性文法に基づいたシステムの多くでは，その記述上の制限から記述が静的な意味に制限されている．このため，副作用の扱いや，インタプリタなどの動的な意味の記述が難しくなっているが，言語指向のプログラミング環境やユーザインターフェイスの記述など対話的な環境は，最近のプログラミング・システムには不可欠であり，言語処理系を記述するためのメタ言語は，こういった制限が取り払われていることが望ましい．実際，プログラミング言語の実行時の実行の履歴や，ユーザとの対話の履歴などは，その実行に対応した実行環境の変化時系列に対応した列と考えるのが自然である．ストリームに基づいた意味記述モデルでは，コンパイラにおけるコード生成など静的な意味の記述から，インタプリタにおける制御構造の記述など動的意味の記述まで，言語指向の応用における様々な局面に適用可能な方式であると考えている．

また，ストリームに基づいた意味記述では，解析木上に定義されたプロセスのストリームによる結合関係により意味記述の定義を行なう．これによって，具象構文の定義と意味記述の分離を明確にしている．意味処理の実現の詳細は，意味処理を行なうプロセス内に隔離される．このため，言語に対する具象構文を変更した場合でもプロセスの結合の定義を変更することにより対応することができる．また，多くの言語には同一のセマンティクスを持つ言語コンストラクトが多く存在する．例えば，while 文のセマンティクスは多くの言語で共通である．こういった複数の言語に共通な言語コンストラクトに対応したプロセスを用意することによって，言語処理系の構成をモジュール化し，その開発を容易にすることができる．また，入出力などをプロセス内に隠蔽することも可能となる．

本論文で示したストリームに基づいた意味処理では，言語処理系における意味記述を，並列な実行を中心として定義するモデルに基づいている．これは，並列な実行を基本とした枠組から出発することによって，将来の高並列なハードウェア環境へのスムースな移行を可能とするためである．現在粒度の小さなプロセスを実行するための方式の研究が勢力的に行なわれている．細粒度プロセスによる実行は問題に内在する並列性を記述し，並列性を抽出するための効率的な方式であるため，近い将来このようなハードウェアが広範囲に利用可能になると予想される．このような点から，言語処理系における並列処理技法の研究は，このような並列環境での効率的なプログラム開発をサポートする礎となると考える．

現在，ストリームに基づいた意味記述モデルは，共有メモリ型のマルチプロセッサシステム上に実現されている．ストリームに基づいた意味記述モデルでは基本的な実行用のハードウェア・アーキテクチャのモデルに対しては特に制約を加えていない．ただし，ストリームに基づいた意味記述モデルを実際に実現し，評価を与えるには何らかのハードウェア・システムを設定する必要がある．5 章で示した意味処理実行用の仮想マシンとマルチプロセッサ・アーキテクチャ SMiS は，現状で構成可能な一般的なハード

ウェア・システム上での並列な意味処理の評価のための実験環境として用意した．本研究の目的は，細粒度プロセスの実行をサポートするハードウェアの研究ではなく，言語処理系における並列処理方式の研究であるため，ハードウェアシステムを固定し，その上に仮想マシンという形でハードウェアに依存しない実行環境を構築している．6章および7章では，SMiS上に構築された仮想マシンで実行を行なうことにより意味記述の評価を行なっている．

ストリームに基づいた意味記述モデルの実現は，解析木のノードに対して定義されたプロセスの，完全に並列な実行形態から開始して，実行モデル中に逐次的な実行を採り入れることによって効率化を行なっている．現状の逐次的な実行を対象としたハードウェアでは，細粒度プロセスによる実行は高いコストを必要とする．また，解析木上に定義されたプロセスと，そこで処理されるデータの依存関係によっては，プロセスの実行がある一定の逐次的な解釈しか許さない場合や，ストリームに基づいた意味処理でのストリームによるデータのネットワーク的な表現に馴染まない場合もある．このため，ストリームに基づいた意味記述の効率的な実行のためには，現状では逐次的な実行形態を，実行モデル中に導入する必要がある．

6章における細粒度実行モデルは，基本的にストリームに基づいた意味記述モデルの直接的な実現であり，ストリームに基づいた意味記述モデルの実現の可能性を探ると共に，その実現上の問題点や，ストリームに基づいた意味記述モデルでの言語処理系記述における効率化手法を研究することを目的として実現されている．また，細粒度実行モデルでは逐次的な処理を導入するための枠組として，解析木上に定義された仮想的なプロセスに対して，これらのプロセスの依存関係に対応した評価グラフを動的に構成し，これを逐次的に評価することによって，並列な実行モデル中に逐次的な処理方式を導入する，プロセスの逐次化と呼ばれる方式を実現した．

以上のような細粒度の実行モデルに基づいて，6章ではPL/0コンパイラの実現を行なった．この実現をもとにして細粒度実行モデルにおける現状での問題点を探ると共に，意味処理を行なうプロセス間のストリームの受渡しによる，意味処理プロセスネットワークの再構成など，記述および実現のための技法の有効性を確認した．言語の構文やデータの構造は，文脈自由文法によって与えることができるが，問題に対する意味的な構造は解析木によって認識される木構造に必ずしも対応するとは限らない．例えば，プログラミング言語におけるラベルとその参照などはこのような例である．このため，意味処理プロセスのネットワーク構造の再構成は，さまざまな対象を記述する上で，重要な手法であると考える．また，6章および7章での実験では，5章で述べたSMiSアーキテクチャのシミュレータ上で評価を行なっている．このため，6章では，キャッシュのヒット率などハードウェアの低レベルのパラメータに関して，実験を行なうことによって実験環境に対しても同時に評価を与えている．

6章で実現した細粒度の実行モデルでは，プロセスの依存関係に対応した評価グラフを構成することによって，逐次的な処理を導入したとはいえ，逐次的な意味の実行に対する実行速度の向上は不十分なものであった．これに対処するのが，7章での疎なプロセス割り当てによる実行モデルである．疎なプロセス割り当てによる実行モデルでは，解析木の認識と同時に逐次的に実行可能な解析木上の領域を構文レベルで指定することにより，低いオーバヘッドでプロセスの動的な統合処理を可能としている．

以上のように，本論文で示すような言語処理系における並列処理の手法は，言語処理系の処理の高速化という面だけでなく，並列システムや分散システム上でのデータに対する操作の適用の並列な記述や，ユーザインターフェイスの記述など様々な対象に適用可能であると考えている．また，ここで示すような，言語処理系における一連の操作を構造を持ったデータに対する意味操作を特殊化したものと捉えることにより，本論文で示した言語処理系における並列実行のためのモデルを構造を持ったデータに対する並列実行，あるいは，分散環境でのデータ操作のための統一的な枠組として用いることなども可能であると考えている．

本論文で示したストリームに基づいた意味処理は，現状では簡単なコンパイラの記述やインタプリタなど，言語処理系の一部の実現に適用を行なったのみである．本手法のより実際的な評価には，様々な具体的な対象に対する実現を行ない，実現技法に関する研究をさらに進める必要があると考えている．

謝辞

本研究を行なうにあたり，筑波大学電子・情報工学系 板野肯三助教授には，研究の開始から現在に至るまで，終始適切な御助言を頂きました．本研究を行なう環境を与えて頂いたことに感謝致します．

本論文の審査を行なって頂いた，同学系の中田育男教授，西原清一教授，清木康助教授，および東京工業大学理学部情報科学科の佐々政孝教授には論文の内容や構成に関して有意義な御指摘を頂きました．感謝致します．

高級言語計算機研究室およびソフトウェア研究室の諸先生方，卒業生および学生の皆さんには，直接かつ間接的に影響を受けると共に，御指導頂きました．感謝します．

妻暁薇は研究仲間として，また共同生活者として力になってくれました，感謝します．娘万里には両親ともに研究に忙しかったため苦労をかけました．最後に感謝したいと思います．

参考文献

[Aho 86] Aho,A.V., Sethi,R. and Ullman,J.D.: Compilers - Principles, Techniques, and Tools, Addison-Wesley (1986).

[Alblas 91a] Alblas,H.: Attribute Evaluation Methods, LNCS-545, pp.48-113, Springer-Verlag (1991).

[Alblas 91b] Alblas,H.: Introduction to Attribute Grammars, LNCS-545, pp.1-15, Springer-Verlag (1991).

[Ambriora 85] Ambriora,V. and Montangero,C.: Automatic Generation of Execution Tools in a GANDALF Environment, The Journal of Systems and Software, Vol.5, pp.155-171 (1985).

[Andre 81] Andre,F., Banatre,J.P. and Routeau,J.P.: A Multiprocessing Approach to Compile-Time Symbol Resolution, ACM Transactions on Programming Languages and Systems, Vol.3, No.1, pp.11-23 (1981).

[Archibald 86] Archibald,J. and Bear,J.L.: Cache Coherence Protocols, ACM Transactions on Computer Systems, Vol.4, No.4, pp.273-298 (1986).

[Baalbergen 88] Baalbergen,E.H.: Design and Implementation of Parallel Make, Computing Systems, Vol.1, No.2, pp.135-158 (1988).

[Banatre 79] Banatre,J.P., Routeau,J.P. and Trilling,L.: An Event Driven Compiling Technique, CACM, Vol.22, No.1, pp.34-42 (1979).

[Bare 77] Bare,J.L. and Ellis,C.S.: Model, Design and Evaluation of a Compiler for a Parallel Processing Environment, IEEE Transactions on Software Engineering, Vol.3, No.6, pp.394-405 (1977).

[Barford 89] Barford,L.A and Vander-Zanden,B.T.: Attribute Grammars in Constraint-Based Graphics Systems, Software-Practice and Experience, Vol.19, No.4, pp.309-328 (1989).

[Bochmann 76] Bochmann,G.V.: Semantic Evaluation from Left to Right, CACM, Vol.19, No.2, pp.55-62 (1976).

[Brown 87] Brown,M.H.: Algorithm Animation, MIT Press (1987).

[Cohen 85] Cohen,J. and Kolodner,S.: Estimating the Speedup in Parallel Parsing, IEEE Transactions on Software Engineering, Vol.11, No.1, pp.114-124 (1985).

[Cypress 90] Cypress Semiconductor: Sparc RISC User's Guide, ROSS Technology,Inc., Cypress Semiconductor Company (1990).

[Deransart 88] Deransart,P., Jordan,M., Lorho,B.: Attribute Grammars, LNCS-323, Springer-Varlag (1988).

[Earley 72] Earley,J., Caizergues,P.: A Method for Incrementally Compiling Languages with Nested Statement Structure, CACM, Vol.15, No.12, pp.1040-1044., (1972).

[Feldman 86] Feldman,S.I.: Make - A Program for Maintaining Computer Programs, Unix Programmers Manual (1986).

[Fischer 75] Fischer,C.N.: Fischer,C.N.: On Parsing Context-Free Languages in Parallel Environments, Ph.D. Dissertation, Department of Computer Science, Cornell Univ. (1975).

[Fischer 88] Fischer,C.N. and LeBlance Jr.,R.J.: Crafting a Compiler, The Benjamin/Cummings Publishing Company,Inc. (1988).

[Friedman 92] Friedman,D.P., Wand,M. and Haynes,C.T.: Essentials of Programming Languages, MIT Press (1992).

[Ganapathi 82] Ganapathi,M.,Fischer,C.N. and Hennessy,J.L.: Retargettable Compiler Code Generation, ACM Computing Survey, Vol.14, No.4, pp.573-592 (1982).

[Graham 80] Graham,S.L.: Table-driven Code Generation, IEEE Computer,Vol.13,No.8, pp.25-34 (1980).

[Gross 89] Gross,T., Zobel,A. and Zolg,M.: Parallel Compilation for a Parallel Machine, In Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, pp.91-100 (1989).

[Hoare 74] Hoare,C.A.R.: Communicating Sequential Processes, CACM, Vol.21, No.8, pp.547-557 (1974).

[Inmos 86] Inmos: Transputer Reference Manual (1986).

[Inmos 88] Inmos: Occam2 Reference Manual,Parentice Hall (1988).

[Itano 88] Itano,K., Sato,Y., Hirai,H. and Yamagata,T.: An Incremental Pattern Matching Algorithm for the Pipelined Lexical Scanner,Information Processing Letter, Vol.27, No.5, pp.253-258 (1988).

[Itano 89] Itano,K., Nishiyama,H. and Chu, Y.: A Bottom-up Parsing Coprocessor for Compilation, Technical Report TR-2280, Department of Computer Science, University of Maryland (1989).

[Johnson 75] Johnson,S.C.: Yacc-Yet Another Compiler Compiler, Computer Science Technical Report, No.32, Bell Laboratories (1975).

[Jones 86] Jones,L.G. and Simon,J.: Hierarchical VLSI design system based on attribute grammars, In Proceedings of the 13th ACM Symposium on Principles of Programming Languages, pp.58-69 (1986).

[Jourdan 91] Jourdan,M.: A Survey of Parallel Attribute Evaluation Methods, in Attribute Grammers, Applications and Systems, Alblas, H. and Melichar,B. eds., LNCS-545, pp.234-255, Springer-Verlag (1991).

[Kaiser 89] Kaiser,G.E.: Incremental Dynamic Semantics for Language-Based Programming Environment, ACM Transactions on Programming Languages and Systems, Vol.11, No.2, pp.169-193 (1989).

[Katayama 85] 片山卓也: 属性文法による在庫管理システムの記述, 情報処理, Vol.26, No.5, pp.478-485.

[Katevenis 84] Katevenis,M.G.H.: Reduced Instruction Set Architecture for VLSI, MIT Press (1984).

[Klein 90] Klein,E. and Koskimies,K.: Parallel One-pass Compilation, LNCS-461, pp.76-90, Springer-Verlag (1990).

[Knuth 69] Knuth,D.E.: Semantics of Context-free Languages, Math. Sys. Th, Vol.2, No.2 (1968).

[Koike 89] 小池稔, 大川善邦: 高水準言語のコンパイル過程の並列処理の研究, 情報処理学会論文誌, Vol.30, No.5, pp.605-611 (1989).

[Koskimies 91] Koskimies,K.: Object-Orientation in Attribute Grammars, LNCS-545, pp.298-329, Springer-Verlag (1991).

[Krohn 75] Krohn,H.E.: A Parallel Approach to Code Generation for Fortran Like Compilers, Proceedings of the ACM Conference on Programming Languages and Compilers for Parallel and Vector Machines, SIGPLAN Notices, Vol.10, No.3, pp.146-152 (1975).

[Kuiper 90] Kuiper,M.F. and Swierstra,S.D.: Parallel Attribute Evaluation: Structure of Evaluators and Detection of Parallelism, LNCS-461, pp.61-75, Springer-Verlag (1990).

[Lesk 75] Lesk,M.E.: Lex - a lexical analyzer generator, Computer Science Technical Report, No.39, Bell Laboratories (1975).

[Liu 88] 劉彭, 清木康, 益田隆司: ストリーム指向型関係演算処理におけるバッファ資源割当ての計算方式, 情報処理学会論文誌, Vol.29, No.8, pp.770-781 (1988).

[Nishiyama 88b] 西山博泰: ハードウェア・コンパイラの設計, 筑波大学情報学類卒業論文 (1988).

[Nishiyama 90a] 西山博泰 : Parallel Compilation based on Streams, 筑波大学大学院工学研究科修士課程論文 (1990).

[Noll 92] Noll,T. and Vogler,H.: Top-Down Parsing with Simultaneous Evaluation of Noncircular Attribute Grammars, Technical-Report 92-14 (1992).

[Pemberton 82] Pemberton,S. and Daniels,M.C.: Pascal Implementation - The P4 Compiler, Ellis Horwood (1982).

[Pennello 86] Pennello,T.J.: Very Fast LR Parsing, Proceedings of SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices, Vol.21, No.7, pp.145-151 (1986).

[Reps 84] Reps,T.W.: Generating Language-Based Environments, MIT Press (1984).

[Sassa 89] 佐々政孝: プログラミング言語処理系, 岩波書店 (1989).

[Sato 86] 佐藤豊: プログラミング支援システムの構成法に関する研究, 筑波大学工学研究科学位論文 (1986).

[Scheifler 86] Scheifler,R.W. and Gettys,J.: The X Windows System, ACM Transactions on Graphics, Special Issues on User Interface Software (1986).

[Seshadri 88] Seshadri,V., Wortman,D.B., Junkin,M.D., Weber,S., Yu,C.P. and Small,I.: Semantic analysis in a concurrent compiler, Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp.233-240 (1988).

[Shapiro 89] Shapiro,E.: The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, Vol.21, No.3, pp.413-510 (1989).

[Shinoda 86] 篠田陽一, 片山卓也: 属性文法による UNIX ファイルシステムの記述, 信学技法 SS86-6, 情報処理学会 SF-86-8, ソフトウェア科学会 PM-86-6 (1986).

[Shinoda 90] Shinoda,Y. and Katayama,T.: Object-oriented Extension of Attribute Grammars and its Implementation Using Distributed Attribute Evaluation Algorithms, LNCS-461, pp.177-191, Springer-Verlag (1990).

[Stallman 86] Stallman,R.: GNU Emacs Manual, Free Software Foundation (1986).

[Stallman 91] Stallman,R.M. and McGrath,R.: GNU Make, Free Software Foundation (1991).

[Stallman 92] Stallman,R.M.: Using and Porting GNU CC, Free Software Foundation (1992).

[Sun 91] Sun Micro Systems: Programming Utilities & Libraries, SunOS Reference Manual, Sun Micro Systems (1991).

[Stoy 77] Stoy,J.E.: Denotational Semantics: The Scott-Strachey Approach To Programming Language Theory, MIT Press (1977).

[Swierstra 91] Swierstra,D. and Vogt,H.: Higher Order Attribute Grammars, LNCS-545, pp.256-296, Springer-Verlag (1991).

[Watt 86] Watt,D.A.: Executable Semantic Descriptions, Software - Practice and Experience, Vol.16, No.1, (1986).

[Wirth 76] Wirth,N.: Algorithms + Data Structures = Programs, Parentice Hall (1976).

[ZYCAD 88] ZYCAD Corporation: N.2 ISP' User's Manual, ZYCAD Corporation (1988).

公表論文

[Ng 88] ウン・チョン・セン, 西山博泰, 板野肯三: ハードウェア LR パーサの構成, 情報処理学会第 36 回全国大会, pp.853-854 (1988).

[Nishiyama 88a] 西山博泰, ウン・チョン・セン, 板野肯三: ハードウェア・コンパイラの設計, 情報処理学会第 36 回全国大会論文集 (1988).

[Nishiyama 89a] 西山博泰, 板野肯三: コンパイラにおける意味処理の並列化, 第 30 回プログラミングシンポジウム報告, pp.123-134 (1989).

[Nishiyama 89b] 西山博泰, 板野肯三: ストリームに基づいた並列意味処理の記述, 情報処理学会プログラミング言語研究会報告 22-3, pp.1-8 (1989).

[Nishiyama 89c] 西山博泰, 板野肯三: ハードウェアコンパイラにおける並列意味解析器の構成, 情報処理学会第 38 回全国大会論文集, pp.909-910 (1989).

[Nishiyama 90b] 西山博泰, 板野肯三: ストリームに基づいた並列意味処理の記述, 情報処理学会論文誌, Vol.31, No.5, pp.731-739 (1990).

[Nishiyama 91] 西山博泰, 板野肯三: マルチプロセッサ・システム SMiS 上での並列コンパイラ Compas の実現と性能評価, 情報処理学会プログラミング - 言語・基礎・実践 - 研究会 3-22, pp.195-203 (1991).

[Nishiyama 92a] 西山博泰, 板野肯三: コンパイラの意味処理の疎粒度並列処理の実現と評価, 情報処理学会プログ ラミング - 言語・基礎・実践 - 研究会 6-6, pp.51-60 (1992).

[Nishiyama 92b] 西山博泰, 板野肯三: 並列コンパイラ Compas の意味処理部の性能評価, 情報処理学会論文誌, Vol.33, No.10 (1992).

[Nishiyama 92c] 西山博泰, 板野肯三: コンパイラの意味処理の疎粒度並列処理の実現と評価, 情報処理学会論文誌, 投稿中 (1992).

付録 A

SSGL の構文定義

SSGL の構文を以下に示す．ここで，| は選択，[ ] は指定した文法記号列が省略可能であることを，{ } は文法記号列の 0 回以上の繰り返しを表す．また，’’で括られた文字列は字句レベルのトークンを表す．

```
<semantic descriptions>
      → <declarations>
        '%%'
          <syntax and semantics definitions>
        { <syntax and semantics definitions> }
        '%%'

<syntax and semantics definitions>
      → <nonterminal symbol>
                ':' { <(non)terminal symbol> } [<semantic rules>]
              { '|' { <(non)terminal symbol> } [<semantic rules>] }
                ';'

<termnal symbol>              → ID | CHAR

<nontermnal symbol>           → ID

<(non)termnal symbol>         → ID | CHAR


<semantic rules>
      → '{' { <semantic rule> } '}'


<semantic rule>
      → <stream> '=' <stream> ';'
       | <stream> '=' <process> ';'
       | '[' <stream> { ',' <stream> } ']' '=' <process> ';'
       | <process> ';'


<stream>
      → <(non)terminal symbol> '.' <stream name>
```

```
      | <(non)terminal symbol> '[' NUM ']' '.' <stream name>

<stream name>
      → ID


<process>
      → ['SEQ'] <process id> '(' [ <arg> { ',' <arg> } ] ')'

<process id>
      → ID

<arg> → <stream> | <constant> | <function>

<function>
      → <function id> '(' [ <arg> { ',' <arg> } ] ')'

<function id>
      → ID

<constant>
      → ID | NUM | STRING | CHAR


<declarations>
      → <type declaration>
       | <include declaration>
       | <token declaration>
       | <operator declaration>


<type declaration>
      → '%' 'atomic' { <stream name> }
       | '%' 'type' <type> ['*'] <stream name>

<type> → ID


<include declaration>
       → '%' 'include' STRING


<token declaration>
      → '%' 'token' <terminal symbol> { <terminal symbol>  }
       | '%' 'token' <terminal symbol> ':' <regular expression>
              [ '{' { <attribute definition> } '}' ]
       | '%' 'skip' <regular expression>
              [ '{' { <attribute definition> } '}' ]

<attribute definition>
      → <terminal symbol> '.' <stream name>
                    '=' <attribute> ';'
       | <attribute function>
```

```
<attribute>
      → (<constant>|<attribute function>|'[' <attribute function> ']'|'@')

<attribute function>
      → <process id> '(' [ <attribute> { ',' <attribute> } ] ')'

<regular expression>
      → STRING

<operator declaration>
      → '%' 'nonassoc' <terminal symbol> { <terminal symbol> }
       | '%' 'left'  <terminal symbol> { <terminal symbol> }
       | '%' 'right' <terminal symbol> { <terminal symbol> }
```

付録 B

SSDL の構文定義

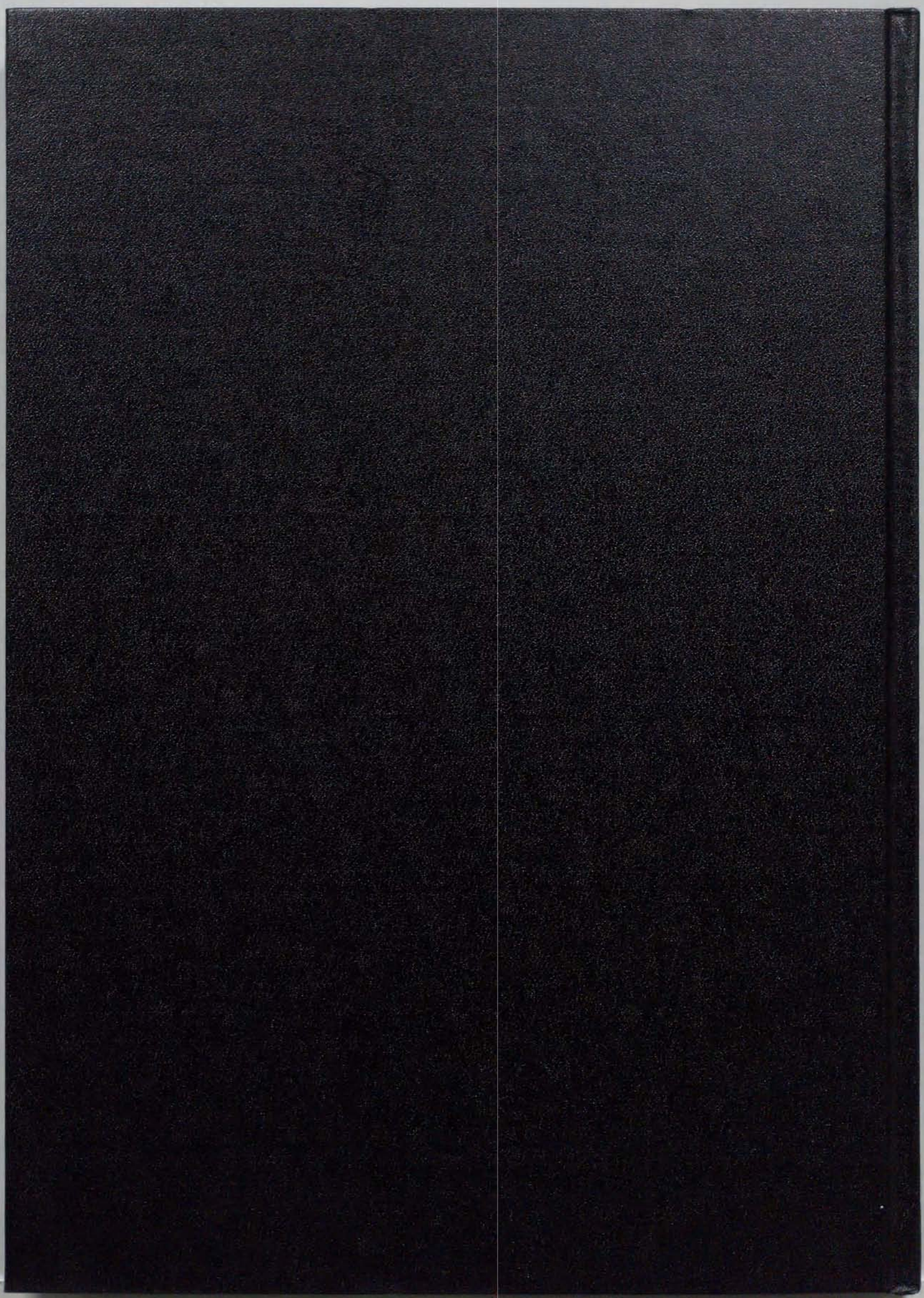
SSDL の構文を以下に示す．ここで，| は選択，[ ] は指定した文法記号列が省略可能であることを，{ } は文法記号列の 0 回以上の繰り返しを表す．また，’’で括られた文字列は字句レベルのトークンを表す．

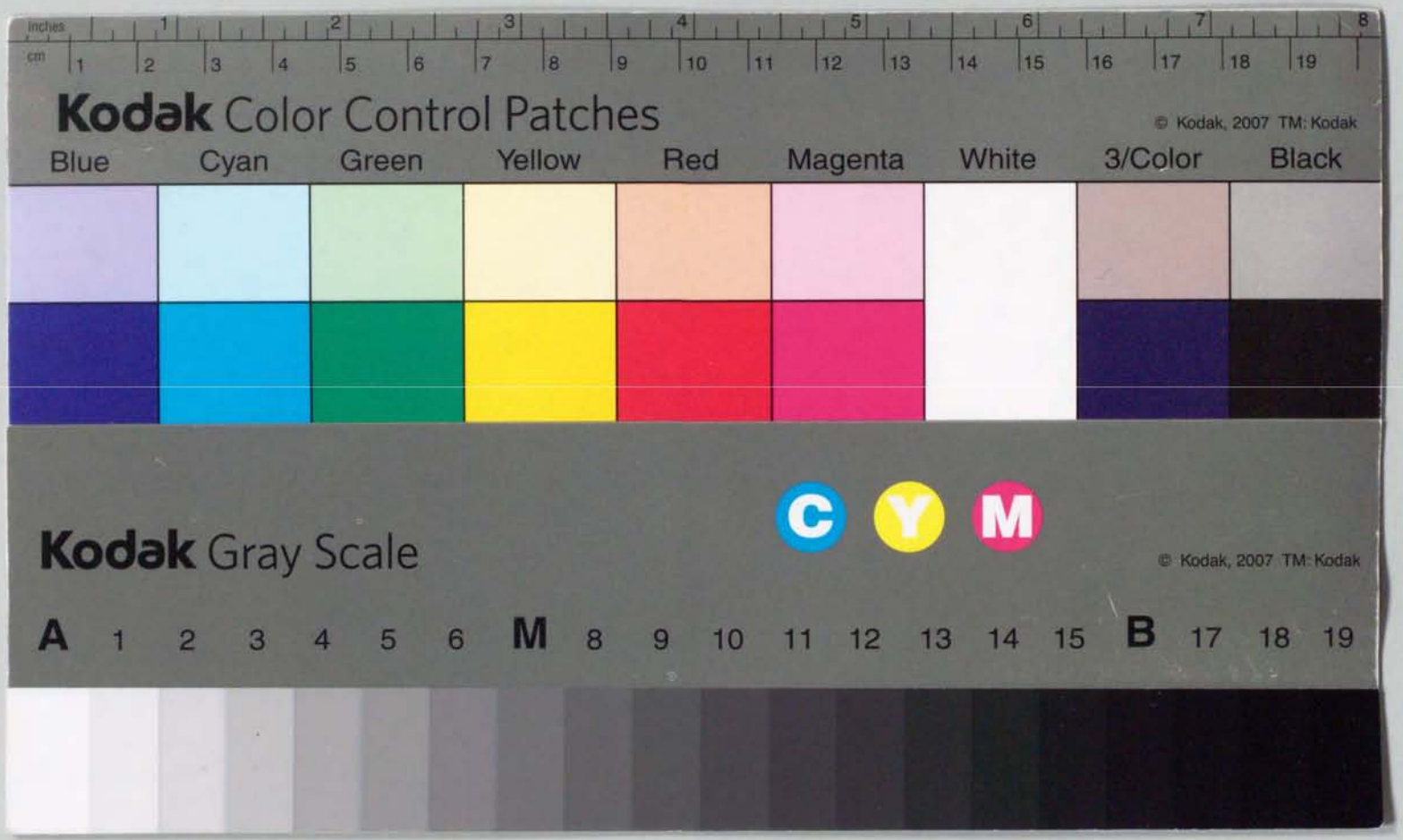
```
<definitions>
      → { (<constant definition>
          |<type declaration>
          |<import declaration>
          |<process definition>
          )
        }

<constant definition>
      → 'CONST' ID '=' <val> { ',' ID '=' <val> } ';'

<val> → NUM | STRING | CHAR| ID

<import declaration>
      → 'IMPORT' STRING ';'

<type declaration>
      → 'TYPE' ID { ',' ID } ';'

<process definition>
      → <process header>
        '{' { <declaration> } { <statement> } '}'

<process header>
      → [ <outputs> '<-' ] ID '(' [ <input> { ',' <input> } ] ')'

<outputs>
      → <stream>
       | '[' <stream> { ',' <stream> } ']'
```

```
<declaration>
      → <type> <variable declaration>
              { ',' <variable declaration> } ';'

<variable declaration>
      → [ '*' ] <variable> [ '=' <exp> ]

<input>
      → <stream>|<type> <variable>

<type>
      → ID


<stream>
      → <stream name> | '@' <stream name>

<stream name>
      → ID


<statement>
      → ';'
       | 'break' ';'
       | 'exit' ';'
       | 'if' '(' <exp> ')' <statement> [ 'else' <statement> ]
       | 'while' '(' <exp> ')' <statement>
       | 'for' '(' [<exp>] ';' [<exp>] ';' [<exp>] ')' <statement>
       | <exp> ';'
       | '{' {<statement>} '}'
       | <select statement>
       | <process statement>


<select statement>
      → 'select' '{' {<selection>} '}'

<selection>
      → [ '(' <exp> ')' '&&' ]
              <stream name> ('->'|'-->') <variable>
                            <statement>
                      [ ':' <statement> ]


<process statement>
      → 'process' process
       | 'process' '{' process { process } '}'

<process>
      → [ <output> '<-' ] ID '(' [<exp> {',' <exp>}] ')'

<output>
      → <stream name>
       | '[' <stream name> ']'
```

```
<exp> → <stream expression>
      | <atomic stream expression>
      | <exp> '+' <exp>        | <exp> '-' <exp>
      | <exp> '*' <exp>        | <exp> '/' <exp>
      | <exp> '%' <exp>
      | <exp> '=' <exp>
      | <exp> '&&' <exp>       | <exp> '||' <exp>
      | <exp> '==' <exp>       | <exp> '!=' <exp>
      | <exp> '>=' <exp>       | <exp> '>'  <exp>
      | <exp> '<=' <exp>       | <exp> '<'  <exp>
      | '++' <exp>             | '--' <exp>
      | <exp> '++'             | <exp> '--'
      | '!' <exp>
      | '+' <exp>              | '-' <exp>
      | '(' <exp> ')'
      | <variable>             | <val>
      | <function>

<function>
     → ID '(' [ <exp> { ',' <exp> } ] ')'

<variable>
     → ID
```

```
<stream expression>
     → <stream name> '<-' <exp>
      | <stream name> '<-' '[' [ <exp> {',' <exp>} ] ']'
      | <stream name> '->' <variable>
      | <stream name> '->' <variable> '<' <stream name> '>'
      | <stream name> '->' '[' ']'


<atomic stream expression>
     → <stream name> '<--' <exp> '-->' <variable>
      | <stream name> '-->' <variable>
      | <stream name> '<--' <exp>
```




inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19