(19)日本国特許庁(JP)　(12)特　許　公　報(B2)　(11)特許番号
特許第3648268号
(P3648268)
(45)発行日　平成17年5月18日(2005.5.18)　(24)登録日　平成17年2月18日(2005.2.18)

(51)Int.Cl.[7]　F I
G06F 9/45　G06F 9/44 322B
G06F 9/44 322L

請求項の数 10　(全 44 頁)

(21)出願番号　特願平6-109864
(22)出願日　平成6年5月24日(1994.5.24)
(65)公開番号　特開平7-141201
(43)公開日　平成7年6月2日(1995.6.2)
審査請求日　平成13年5月18日(2001.5.18)
(31)優先権主張番号　066687
(32)優先日　平成5年5月24日(1993.5.24)
(33)優先権主張国　米国(US)

(73)特許権者　398038580
ヒューレット・パッカード・カンパニー
HEWLETT-PACKARD COMPANY
アメリカ合衆国カリフォルニア州パロアルト　ハノーバー・ストリート　3000
(74)代理人　100063897
弁理士　古谷　馨
(74)代理人　100076680
弁理士　溝部　孝彦
(74)代理人　100087642
弁理士　古谷　聡

最終頁に続く

(54)【発明の名称】２パスコンパイラのための改良された方法

(57)【特許請求の範囲】
【請求項１】
第１のコンピュータ言語を第２のコンピュータ言語にコンパイルするためのコンピュータにより実施される方法であって、前記コンピュータがプロセッサおよび記憶装置を含み、前記コンパイルが少なくとも２回のコンパイルパスを必要とするものにおいて、前記コンピュータにより実施される方法が、
前記第１のコンピュータ言語を前記第２のコンピュータ言語にコンパイルするためのパーサ状態の複数の定義を含むと共に、前記パーサ状態の少なくとも１つにおいて実行するための少なくとも１つのプロダクション関数を含むコンパイラ定義ファイルを前記記憶装置から読み出すステップであって、前記少なくとも１つのプロダクション関数のそれぞれが、第１のコンパイルパスまたは第２のコンパイルパスあるいは第１のコンパイルパスと第２のコンパイルパスの両方で実行されるように指定されている、コンパイラ定義ファイルを前記記憶装置から読み出すステップと、
前記コンパイラ定義ファイルを前記プロセッサによって処理して、前記第１のコンピュータ言語をコンパイルし、及び前記第２のコンピュータ言語を生成するステップとからなり、該ステップが、
前記第１のコンピュータ言語のステートメントを含むソースファイルを読み出し、前記ソースファイルの前記ステートメントの各々に含まれる各記号毎に１つずつ、複数のトークンを生成するステップと、
前記トークンを構文解析して、前記複数のパーサ状態の何れが発生したかを決定し、

前記構文解析によって前記パーサ状態の各々の発生順序を規定するステップと、
前記第１のコンパイルパスにおいて前記複数のパーサ状態の各々の発生時に実行されることを指定された前記プロダクション関数を全て実行し、前記コンパイルパスの両方において前記複数のパーサ状態の各々の発生時に実行されることを指定された前記プロダクション関数を全て実行するステップと、
前記パーサ状態を前記記憶装置のデータ記憶領域へと前記発生順に書き込むステップとからなる第１のコンパイルパスを実行するステップと、
前記記憶装置の前記データ記憶領域から前記パーサ状態を前記発生順に読み出すステップと、
前記第２のコンパイルパスにおいて前記パーサ状態の各々の発生時に実行されることを指定された前記プロダクション関数を全て実行し、前記コンパイルパスの両方において前記複数のパーサ状態の各々の発生時に実行されることを指定された前記プロダクション関数を全て実行し、前記プロダクション関数によって前記第２のコンパイルパスを実行するステップ
とからなる第２のコンパイルパスを実行するステップと
からなる、コンピュータにより実施される方法。

【請求項２】
前記プロダクション関数の各々が、前記第１のコンパイルパス中に実行されるよう指定され、又は前記第２のコンパイルパス中に実行されるよう指定され、或いは前記第１のコンパイルパスと前記第２のコンパイルパスの両方で実行されるよう指定される、請求項１に記載のコンピュータにより実施される方法。

【請求項３】
前記第１のコンパイルパス中に実行される前記プロダクション関数の少なくとも１つの実行時にデータをセーブするステップと、前記第２のコンパイルパス中の前記プロダクション関数の実行時に前記セーブされたデータを復元するステップとをさらに含み、前記データのセーブ及び復元が、前記各パスにおいて前記パーサ状態が発生する前記順序における同じ位置で行われる、請求項１に記載のコンピュータにより実施される方法。

【請求項４】
前記データの前記セーブ及び復元が、前記パーサ状態の各々の発生の間に行われる、請求項３に記載のコンピュータにより実施される方法。

【請求項５】
前記データの前記セーブ及び復元が、前記コンパイルパスの各々において所定のプロダクション関数が実行される際に行われる、請求項３に記載のコンピュータにより実施される方法。

【請求項６】
前記第２のコンパイルパスにおける実行に備えて、前記第１のコンパイルパスにおいて少なくとも１つのプロダクション関数を識別するステップをさらに含む、請求項１に記載のコンピュータにより実施される方法。

【請求項７】
前記第１のコンパイルパス中に前記少なくとも１つのプロダクション関数に渡すべきパラメータデータをセーブし、前記第２のコンパイルパスにおいて前記少なくとも１つのプロダクション関数を実行する際に該少なくとも１つのプロダクション関数に前記パラメータデータを渡すステップをさらに含む、請求項６に記載のコンピュータにより実施される方法。

【請求項８】
前記第１のコンパイルパス中に所定のプロダクション関数を実行する場合に、前記第１のコンパイルパスの処理を中断し、前記第２のコンパイルパスを実行して前記データ記憶領域に記憶された前記パーサ状態の全てを前記第２のコンパイルパスで処理し、前記第１のパスの処理を再開するステップをさらに含む、請求項１に記載のコンピュータにより実施される方法。

【請求項９】
第１のコンピュータ言語を第２のコンピュータ言語にコンパイルするためのコンピュータにより実施される方法であって、前記コンピュータがプロセッサおよび記憶装置を含み、前記コンパイルが少なくとも２回のコンパイルパスを必要とするものにおいて、前記コンピュータにより実施される方法が、
前記第１のコンピュータ言語を前記第２のコンピュータ言語にコンパイルするためのパーサ状態の複数の定義を含むと共に、前記パーサ状態の少なくとも１つにおいて実行するための少なくとも１つのプロダクション関数を含むコンパイラ定義ファイルを前記記憶装置から読み出すステップと、
前記コンパイラ定義ファイルを前記プロセッサによって処理して、前記第１のコンピュータ言語をコンパイルし、及び前記第２のコンピュータ言語を生成するステップとからなり、該ステップが、
前記記憶装置から前記第１のコンピュータ言語のステートメントを含むソースファイルを読み出し、前記ソースファイルの前記ステートメントの各々に含まれる各記号毎に１つずつ、複数のトークンを生成するステップと、
前記トークンを構文解析して、前記トークンについて前記複数のパーサ状態の何れが発生したかを決定し、前記構文解析によって前記パーサ状態の各々の発生順序を規定し、前記複数のパーサ状態の各々の発生時に実行するよう識別された前記プロダクション関数を全て実行するステップと、
前記パーサ状態を前記記憶装置のデータ記憶領域へと前記発生順に書き込むステップとからなるスキャンコンパイルパスを実行するステップと、
前記データ記憶領域から前記パーサ状態を前記発生順に読み出すステップと、
前記パーサ状態の各々の発生時に実行するよう識別された前記プロダクション関数を全て実行し、前記プロダクション関数によって前記第２のコンピュータ言語を生成するステップ
とからなる少なくとも１つのパスを前記スキャンコンパイルパスの後に続けて実行するステップと
からなる、コンピュータにより実施される方法。
【請求項１０】
前記スキャンコンパイルパス時に実行される前記プロダクション関数の少なくとも１つの実行時にデータをセーブし、前記スキャンコンパイルパスに後続する前記少なくとも１つのパスにおいて前記プロダクション関数の実行時に前記セーブされたデータを復元するステップをさらに含み、前記データが、前記スキャンコンパイルパスにおいて前記パーサ状態が発生する前記順序における第１の位置にセーブされ、前記スキャンコンパイルパスに後続する前記少なくとも１つのパスにおいて前記パーサ状態が発生する前記順序における同じ位置で復元され、及び、
第１の所定のプロダクション関数が前記スキャンコンパイルパスにおいて実行される際に前記セーブ位置が識別され、第２の所定のプロダクション関数が前記スキャンコンパイルパスにおいて実行される際に前記データがセーブされる、請求項９に記載のコンピュータにより実施される方法。
【発明の詳細な説明】
【０００１】
【産業上の利用分野】
本発明は、コンピュータシステムに関し、特に、コンピュータシステム内の言語コンパイラに関し、とりわけ、高効率の２パス言語コンパイラの生成方法に関する。
【０００２】
【従来の技術】
コンパイラは、デジタルコンピュータで実行されて或る言語を別の言語に変換するプログラムである。一般に、変換対象となるソース言語と呼ばれる言語は、プログラミング技術における当業者であるコンピュータプログラマが容易に読み取り理解することができる言

語である。ソース言語は一般に、オブジェクト言語へと翻訳される。このオブジェクト言語は、そのコンパイラ自体が実行されるコンピュータのためのマシン語である場合が多い。オブジェクト言語またはマシン語は、一般にコンピュータ自体にしか理解できないものである。

【０００３】

コンパイラは、一般にソース言語の各ステートメントを読み出し、そのステートメントを１つ以上のオブジェクト言語ステートメントへと翻訳する。言語によっては、「前方参照(forward reference)」が可能であり、その場合には、ソースファイル中の或る位置における言語ステートメントが、そのソースファイル中の後続ポイントで発生する情報を参照することになる。ソースファイル中の後続ポイントで情報が発生する場合、コンパイラは、ソース言語のステートメントを１回読み出しただけではその翻訳を完成することができない。これは、前方参照がどこに位置しているかコンパイラが知らないからである。従って、前方参照が可能な場合には、コンパイラは、マシンコードまたは後で最適化される中間コード等のオブジェクト言語へとソースコードを完全にコンパイルするために、ソースコードに対して２回以上のパスを実行しなければならない。

【０００４】

コンパイラは、字句スキャナおよびパーサと呼ばれる２つの主要セクションを用いて構成されることが多い。字句スキャナは、単にスキャナと呼ばれることが多く、ソースステートメントを読み出して、そのソースステートメントからの各ワード記号または数を「トークン」へと分割する。その各トークンには参照記号（数である場合が多い）が付与され、この参照記号がコンパイラのパーサセクションに送られる。コンパイラのパーサセクションは、多数のトークンに処理を行って、それらトークンをその言語で有効なステートメントへと分割する。即ち、スキャナがソースコードをトークンのストリームへと分割し、パーサが、その多数のトークンを解析して、そのトークンがその言語内のステートメントと一致するか否かを判定する。パーサが、１つ以上のトークンと言語内における有効なステートメントとの間に一致を見つけると、パーサ動作ルーチンが呼び出されて、その言語のステートメントに対応するオブジェクト言語コードが出力されることになる。

【０００５】

コンパイラに関する多数パスは、多様な形態をとることが可能なものである。その一形態は、第１パス中にソース言語のスキャンおよび構文解析を行って、そのソース言語中の変数に関する必要とされるタイプ情報を求め、次いでそのタイプ情報を用いて第２パス中に再びソース言語の再スキャンおよび再構文解析を行って、オブジェクト言語コードを実際に生成する。従って、第１パスによって全情報の位置が決定され、これにより、第２パス中には全ての前方参照が既知のものとなって、正確なオブジェクト言語コードを生成することが可能になる。この技法を利用して既存の１パスコンパイラをマルチパスコンパイラに変更することは比較的容易ではあるが、この方法は、ソースコードのスキャンおよび構文解析を２回行うことを必要とし、このため、ソース言語コードをオブジェクト言語コードにコンパイルするのに必要な時間が大幅に増加することになる。また、別の方法では、ソースコードのスキャンおよび構文解析を１回実行して、パーズツリーだけしか用いない第２パスに備えて必要なタイプ情報およびパーズツリーをセーブする。この技法もまた、良好に機能するものではあるが、一般にコンパイラ全体の基本設計に影響を与えることになり、例えば、第２パスのために構文解析規則を別個に規定する必要が生じ、このため、既存のコンパイラを１パスから２パスに改良する場合には困難なプロセスを伴うことになる。

【０００６】

ほとんどの場合、ソースコードをオブジェクト言語コードにうまく変換するために必要なのは２パスだけである。

【０００７】

従って、ソース言語の再スキャンおよび再構文解析を行うことのない効率的な２パスコンパイラが、当業界で必要とされている。更に、旧式の１パスコンパイラを２パスコンパイ

ラに改良する能力を備えたコンパイラライタにとって理解し易いコンパイラが、当業界で必要とされている。更に、両方のパスに関するコードを共にコンパイラのソースコード中に保持する２パスコンパイラが、当業界で必要とされている。

【０００８】

【発明が解決しようとする課題】

本発明の目的は、コンピュータシステム用の２パス言語コンパイラを作成する方法を提供することにある。

【０００９】

本発明の別の目的は、ソース入力が１回しか処理されないコンパイラを作成することにある。

【００１０】

本発明の別の目的は、コンパイラの状態を収容した中間ファイルを利用することにある。

【００１１】

本発明の更に別の目的は、両方のパスについての構文解析命令を単一位置で提供することを可能にすることにある。

【００１２】

【課題を解決するための手段】

本発明の上述その他の目的は、入力ソースコードのスキャンおよび構文解析を第１パス中に行い、再生すべきパーサ状態を第２パス中にセーブするシステムで達成される。従って、第２パス中には、ソースファイルの再スキャンも再構文解析も行われない。パーサ状態が第２パス中に再生され、その第２パス中に文法中の全てのパーサ動作が、第１パスにおける実際の構文解析中に当初に記録されたのと同じ順序で実質的に再実行される。これにより、元の入力言語の再スキャンおよび再構文解析を行う必要がなくなり、コンパイラの処理時間が短縮される。

【００１３】

ユーザは、各文法動作において、各パス中にどのコードを実行すべきかを文法で指定する。このシステムは、文法内および文法外の両方で、２つのパスの間の効率的な連絡のための便宜を提供する。更に、文法外のコンパイラソースコードのために、手順の呼び出しを据え置き、変数値をセーブして自動的に復元し、コンパイラのエラーメッセージを容易に取り扱うための便宜も提供するものである。これらにより、既存の１パスコンパイラの迅速で容易な改変が可能となる。

【００１４】

本発明の上述その他の目的、特徴、および利点は、図面に関連して進める以下の詳細な説明を参照することにより、一層良好に理解されよう。

【００１５】

【実施例】

以下の説明は、現在企図する本発明の最良の実施態様に関するものである。本説明は、制限的な意味に解釈されるべきものではなく、単に本発明の一般的原理を説明することを目的としたものである。本発明の範囲は、特許請求の範囲により判断されるべきである。

【００１６】

本発明は、２つのユーティリティプログラムlex,yaccを利用するものであり、それらは、全てのUNIX（UNIXはAT&Tの登録商標）オペレーティングシステム上で利用可能なものである。lexスキャナおよびyaccパーサジェネレータは、1970年代にベル電話研究所で開発されたものである。スキャナユーティリティlexについては、Alfred V. Aho、Ravi SethiおよびJeffery D. Ullmanによる「Compilers Principles, Techniques, and Tools」(Addison Wesley, p.105～, 1986)に解説されている。yaccパーサジェネレータについては、上記と同じ参考文献の257ページ以降で解説されている。更に、lexおよびyaccの双方については、Tony MasonおよびDoug Brownによる「Lex & Yacc」(O'Reilly & Associates, Inc.,1990)と題する文献に解説されている。

【００１７】

２パスコンパイラを作成するために本発明に入力される言語文法は、追加の２パス指示語を有する標準的なyacc文法である。この文法での各パーサ動作の実行は、後述する文法中の規定により制限されない限り、正確に２回実行される。２つの各パスにおいてパーサ動作を実行させる制御、およびその２つのパス間での連絡は、後述の指示語によって容易化される。

【００１８】

コンパイルすべき言語に関する文法を作成するプログラマは、２つのパスのそれぞれについて各パーサ動作においてどのコードを実行すべきかを指定する。この指定は、パススコープを用いて行われる。それらの特殊なパススコープは、文法内で通常のスコープを使用することができる部分であればどこでも使用することができ、更に、そのスコープ内に含まれるコードを実行すべきパスがどれであるかを指定する。例えば、下記の２パスパーサ動作について考える。

【００１９】

```
defname: SNAME
{
    pass1{$$=def_name($1);}
};
```

この例の場合、def_name()は第１パスにおいてのみ呼び出される。第２パス中には、このパーサ動作は何もしない。利用可能なパススコープは下記の通りである。

【００２０】

```
pass1{/*パス１中でのみ実行すべきコード*/}
      または  1{}
pass2{/*パス２中でのみ実行すべきコード*/}
      または  2{}
pass12{/*両方のパス中で実行すべきコード*/}
      または  *{}
```

従って、上記例は次のように書くこともできた：

```
defname: SNAME
1{
    $$=def_name($1);
};
```

更なる例示のため、下記にもう１つの例を示す。この例の場合、printfの呼び出しが各パスで１回だけ実行される。

【００２１】

```
printpass:    /*empty/*
*{
    1{printf("第１パス中＼n");}
    2{printf("第２パス中＼n");}
}
```

第１パスで作業の一部を行い、第２パスでその作業を続行するのが望ましい場合が極めて

多い。また、字句スキャナは、第１パス中にソースコードのスキャンしか行わないので、第２パス中には、そのスキャナにより生成されるトークンと、恐らくは字句スキャナの何らかの状態とが必要とされる可能性がある。

【００２２】

TWOPASS_SAVE()と呼ばれる関数は、パーサ動作における第１パスコードの後に値をセーブし、次いで、それと同じパーサ動作における第２パスコードの実行中にそのセーブ値を復元する手段を提供する。コンパイラが第１パスを実行する際に、関数TWOPASS_SAVEにより識別された変数が、そのTWOPASS_SAVEの呼び出しが発生したポイントでセーブされ、コンパイラが第２パスを実行する際に、その変数がやはりTWOPASS_SAVEの呼び出しが発生したポイントで復元され、これにより、その変数が第２パス中で利用可能となるようになっている。

【００２３】

例えば、下記のコードの場合には、SICONが第１パス中にキャラクタのポインタを返し、何らかの理由で追加操作を第２パスまで据え置く必要があるものと仮定する。

【００２４】

```
add: SICON '+' SICON
*{
    int a,b;
    1{
        a=atoi($1);/*aはその値をパス１で得る*/
        b=atoi($3);/*bはその値をパス１で得る*/
    }
    TWOPASS_SAVE(a); /*aはパス１でセーブされ、パス２で復元される*/
    TWOPASS_SAVE(b); /*bはパス１でセーブされ、パス２で復元される*/
    2{
        $$=a+b;/*aおよびbはパス２で最終的に使用される/*
    }
};
```

第１パス中で、「a」および「b」は、関数ATOIへの関数呼び出しから返された数値を得る。次いで、それらの変数の値が関数TWOPASS_SAVEの呼び出しによってセーブされてパーサ動作が完了する。第２パス中に、aおよびbの値がTWOPASS_SAVEの呼び出し位置で復元され、それらの値が合計されて、第２パススコープに含まれるコード中の変数$$が生成される。これらの値は、第１パス中でそれらの値が発生した位置と同じトークンストリーム中の位置で第２パス中に復元される。例えば、上記のパーサ動作は、第１パス中に多数回にわたって実行することが可能であり、それに対応する第２パス中での各実行により、前記第１パス中でのコードの実行が発生する毎に、その位置に格納されている値が復元される。

【００２５】

関数TWOPASS_SAVEは、両方のパス中で実行されなければならないので、パススコープ12{}中に含まれなければならない。これは、上記の例の場合にはTWOPASS_SAVEが、*{ で始まるスコープ中にあるので、満たされている。

【００２６】

文法規則外で行われる関数については、付加的な能力が提供される。変数「yypass」は、現在のパス番号（１または２）を含む整数型の変数である。yyparse()がまだ呼び出されていないかまたは終了している場合には、変数yypassの値はゼロとなる。

【００２７】
コンパイラは、第１パス中で関数呼び出しを規定する能力を有しているが、第２パス中でのトークンストリームの再生において、そのトークンが第１パス中でパーサにより当初に処理された場合と同じ位置まで、関数の実際の呼び出しを据え置く。この関数は、twopass_callfunc()と呼ばれ、第１パス中のtwopass_callfunc()の位置と同じ入力トークンストリーム中のポイントで、第２パス中に或る関数が呼び出されるようにする。これは、文法を直接関連させることなく作業を据え置くのに極めて有用な関数である。
【００２８】
例えば、第１パス中および第２パス中にユーザエラーメッセージを生成するコンパイラについて考察する。そのエラーメッセージは、入力言語に基づいて順に現れるのが望ましい（即ち、行nに関する全てのエラーは、行n+1に関するエラーの前にプリントされる）。以下の関数は、この場合を取り扱ったものである。
【００２９】

```
void error(char *str, int line)
{
   if(yypass==1)
   {
      twopass_callfunc(error, "%s%d" , str, line);
      return;
   }
   fprintf(stderr, "Compiler Error, Line %d: %s\n", line, str);
}
```

第１パス中でerror()が呼び出される場合、twopass_callfunc()を呼び出してそれと同じ位置で第２パス中でエラー関数が呼び出されて、fprintステートメントがソース中の同じポイントで第２パス中にのみ実際に実行されるようになる。error()が第２パス中で呼び出される場合には、その関数は単にfprintfを呼び出すことになる。
【００３０】
twopass_callfuncに対するパラメータには、呼び出すべき関数の名称（Ｃプログラミング言語の場合には、関数名の代わりに、その関数に対するポインタが用いられる）と、その関数が実際に呼び出された際にその関数に渡される引数とが含まれる。
【００３１】
関数呼び出しの据え置きと同様の方法により、変数内容を、第１パス中で入力トークンストリーム中の位置に有していた値と同じ値へと第２パス中で復元することが可能である。これは、twopass_restoreと呼ばれる関数を用いて行われる。twopass_restoreは、第１パス中で呼び出された際に変数の内容をセーブする。その変数内容は、第２パス中での入力トークンストリームの再生時と同じ位置で第２パス中に復元される（以下の図１８に関する説明を参照のこと）。twopass_restoreは、２つのパラメータを受容する。その第１パラメータは変数のポインタであり、第２パラメータは変数のバイトで表したサイズである。twopass_restoreは局所変数に利用することはできない。これは、局所変数のアドレスがパス毎に変化することがあるからである。
【００３２】
例えば、構文解析が行われる行の行数を含む変数のように、変数によっては、パス１とパス２との両方について常に同じ値を有していることが必要な場合がある。コンパイラは、最新の状態に保つ必要のあるこれら変数を取り扱うための機構を提供する。この機構は、「TWOPASS_VAR」と呼ばれ、twopass_restoreより一層効率の良いものである。これは、TW

OPASS_VARが、自動的にデータをセーブして復元するからであり、これに対し、関数twopass_restoreは、自動的に復元するだけであり、ユーザは依然としてセーブのための関数を呼び出す必要がある。機構TWOPASS_VARは、以下のように、Ｃプログラミング言語によるプリプロセッサのステートメント「#define」により実施される。

【００３３】

```
#define TWOPASS_VAR1 lineno
#define TWOPASS_VAR1_TYPE int
```

上記２つのステートメントの場合、例えば、コンパイラは、２つのパス間でlinenoを維持する。即ち、それがパス１で変化すると、パス１で変化した各位置においてパス２でも変化することになる。この機構は、好適実施例の場合には、３つの変数に限定され、それらは文法のヘッダ部でのみ定義することができる。

【００３４】

通常は、ファイルの終わりが見つかったことを示すEOFトークンおよびソースコードがスキャナにより返された際に、第１パスが終了して、第２パスが開始される。しかし、任意のポイントで、コンパイラに対し、パス１の処理を強制的に中止させて、直ちにパス２の処理を開始させることが可能である。指示語「TWOPASS_DOPASS2」は、第１パスの終了および第２パスの開始を直ちに発生させる。第２パスが完了すると、スキャナからの新しいトークンを用いて第１パスが再開する。即ち、第１パスが単に中断される一方、第２パスが入力トークンストリーム中のその同じポイントまで処理される。これにより、使用されるダイナミックメモリおよび中間ファイルサイズの縮小が可能となる。恐らくは、第２パスの完了後に、手順記号テーブル等の、指示語までの情報を排除することができるので、メモリの使用量が減少し、また第１パスでの中間ファイルの再開始以降は中間ファイルが小さくなる。

【００３５】

致命的なエラーを検出し、その回復が不可能である場合には、コンパイラは、一般にそのポイントで動作を停止する。しかし、上述の据え置かれた関数の呼び出しのためにエラーメッセージがプリントされなかった可能性があるので、コンパイラは、その実行を停止する前に全てのエラーメッセージをプリントする機構を提供する。この関数は、twopass_error_callfunc()と呼ばれる。この関数は、その呼び出し時に、待ち行列に入っている全てのtwopass_callfunc()をさかのぼって、エラーに関連するものを呼び出し、エラーメッセージがプリント出力されることになる。この関数の呼び出しの後、コンパイラは、その回復不可能なエラーに起因して実行を停止することができる。

【００３６】

このシステムはまた、各パスの開始時に呼び出される２つの関数を提供して、各パスの開始時に行う必要のある初期設定を可能にするものである。

【００３７】

図１は、本発明を組み込んだコンピュータシステムを示すブロック図である。同図のコンピュータシステム100は処理要素102を備えている。この処理素子102は、システムバス104を介してコンピュータシステム100の他の要素との通信を行うものである。キーボード106は、このコンピュータシステムのユーザによるこのシステムへのテキスト入力を可能にし、またマウス110は、このコンピュータシステムへのグラフィックデータ入力を可能にする。グラフィックディスプレイ108は、このシステムのユーザに対してテキスト情報およびグラフィック情報の表示を行うことを可能にする。ディスク112は、本発明のソフトウェアおよびデータファイルを格納するものである。

【００３８】

メモリ116は、オペレーティングシステム118を備えたものであり、このオペレーティングシステム118により、本発明のソフトウェアがこのシステムの様々な入出力装置との通信を行うことが可能になる。プリ／ポストプロセッサ120およびパーサジェネレータ122が使用されて、ディスク112から言語の文法情報が読み出され、オブジェクト言語の言語コマンドに処理を加えてこのコンピュータシステムのユーザのためのコードを生成する２パス

コンパイラ126が生成される。

【００３９】

図２は、本発明の２パスコンパイラを生成するためのプロセスを示すブロック図である。同図において、ディスク112（図１参照）のファイルに含まれている言語文法情報204はプリプロセッサ120に入力される。プリプロセッサ120は、その言語文法を、yaccパーサジェネレータにとって受容可能な形態になるように変更する。次いでその情報がファイル206としてyaccパーサジェネレータ122に送られる。yaccパーサジェネレータ122の出力が、ファイル208であり、ポストプロセッサソフトウェア124に送られる。ポストプロセッサソフトウェア124は、２パスパーサドライバファイル210を読み出し、そのファイルとyaccパーサジェネレータ122の出力208とを組み合わせて、２パスコンパイラのソースコード212を生成する。このソースコードが言語コンパイラ214に入力され、その言語コンパイラ214が、そのソースコードをコンピュータシステム100内で実行可能なオブジェクトコードへと変換する。次いでそのオブジェクトコードが２パスコンパイラ126としてディスク112に格納される。

【００４０】

図３は、本発明の２パスコンパイラの動作を示すブロック図である。同図において、ディスク112上にあるソースコードファイル302からのデータは、字句スキャナ304に入力される。字句スキャナ304は、第１パスパーサ306により呼び出され、これをライン314で示すこととする。字句スキャナは、ソースファイル302からの各ワードや数等を分離して、その各々を、ライン316を介した第１パスパーサ306への「トークン」として識別する。更に、そのトークンが、定義可能な値を有する数またはキャラクタストリング等である場合、その値は、yytextと呼ばれる変数で、ライン318を介して第１パスパーサ306へ送られる。第１パスパーサ306は、字句スキャナ304からのトークンを読み、そのソースコードファイル302から受け取ったトークンに対して言語の文法の突き合わせを行う。その第１パスパーサ306の各状態は、第２パスパーサ308による使用に備えてディスク112の中間ファイル312に格納される。第１パスパーサ306および第２パスパーサ308は両方とも、パーサ動作関数310を呼び出す。このパーサ動作関数310は、２パスコンパイラ126の生成時に言語文法204（図２参照）内で定義されるものである。第２パスパーサ308の出力324は、コンピュータシステム100内での実行に適したコンパイルされたコードである。

【００４１】

図４は、図２のプリプロセッサブロック120およびポストプロセッサブロック124を示すブロック図である。図２に別個に示すように、プリプロセッサ120およびポストプロセッサ124は、好適実施例では単一のモジュールとして実施される。この単一モジュールは、言語文法204の前処理を行って中間ファイル410を生成する。次いでプリ／ポストプロセッサ406は、子プロセスを生成してその子プロセス中でyaccパーサジェネレータ408を実行することによりyaccパーサジェネレータ408を呼び出し、そのyaccパーサジェネレータ408に入力として中間ファイル410を送る。yaccパーサジェネレータ408は、その中間ファイル410の処理を行って、パーサ出力ファイル412を生成する。yaccパーサジェネレータ408がその処理を完了すると、プリ／ポストプロセッサパーサ406に戻り、そのプリ／ポストプロセッサパーサ406が、パーサ出力ファイル412を読み出してその処理を行ってコンパイラ出力ソースファイル212を生成し、次いでそれをコンパイラ214（図２参照）に送る。

【００４２】

図５ないし図７は、図４のプリ／ポストプロセッサパーサ406を示すフローチャートである。図５のブロック502でプリ／ポストプロセッサパーサに入る。ブロック502に入ると、同ブロックで、入力コマンド行で送られた最初のまたは次のパラメータを得る。ブロック504では、そのパラメータがシーケンス「-tp:」で始まるか否かを判定し、シーケンス「-tp:」で始まる場合には、ブロック506に移行して、２パスコンパイラのためにそのパラメータの処理を行う。また、そのパラメータがシーケンス「-tp:」で始まらない場合には、ブロック504からブロック508に移行して、yaccに送るべきパラメータをセーブする。パラメータの処理またはパラメータのセーブの後に、制御がブロック502に移行して、コマン

ド行のパラメータが更に存在するか否かを判定し、存在する場合には、ブロック502に戻って次のパラメータの処理を行う。

【００４３】

全てのコマンド行パラメータの処理が完了した後、ブロック510からブロック512に移行して、入力ファイルおよび出力ファイルをオープンし、次いで、ブロック514で、図８の関数yyparseを呼び出して、入力ファイルに処理を行って図４の中間ファイル410にする。入力ファイルの処理が完了した後、関数yyparseからブロック516に戻ってファイルをクローズし、次いでブロック518で、yaccプロセッサを実行して、中間ファイル410がyaccパーサジェネレータに入力ファイルとして与えられる。このyaccパーサジェネレータは、図４にブロック408として示されている。好適実施例では、yaccパーサジェネレータは、子プロセスとして実行されるが、関数として呼び出すことも可能である。yaccの動作は、Ahoによる参考文献およびMasonによる参考文献に記載のように当業界で周知のものであり、更なる説明は省略することとする。

【００４４】

yaccパーサジェネレータ408（図４参照）が処理を完了した後にパーサ出力ファイル412が生成され、このファイルは、プリ／ポストプロセッサパーサ406によって後処理が行わなければならない。この後処理を行うために、ブロック518からブロック602に移行し、入力としてyacc出力ファイル412をオープンする。次いでブロック604で、入力のために２パスドライバファイル210（図２参照）をオープンし、ブロック606で、図２および図４の出力ファイル212をオープンする。次いで制御がブロック608に移行して、入力ファイル412から次の行を得る。次いでブロック610で、入力ファイル全体の処理が完了したか否かを判定し、完了していない場合には、ブロック612に移行して、読み出したばかりの行がドライバファイルの最初の行と同じであるか否かを判定する。それら行が同じである場合には、ブロック612からブロック614に移行して、シーケンス「@@@@」が生じるまでドライバファイルの各行を出力ファイルにコピーする。それらの行のコピーが完了した後、ブロック616で、入力ファイルからパーサコードを廃棄する。こうして、２パスパーサドライバファイル210は、パーサ出力ファイル412中にあるパーサドライバの関数yyparse()をyaccパーサジェネレータ408に置き換える。入力ファイル412からパーサコードの関数yyparseを廃棄した後、ブロック616からブロック608に戻って、入力ファイルから次の行を得る。

【００４５】

また、読み出したばかりの行がドライバファイルの最初の行と一致しない場合には、ブロック612からブロック618に移行し、その行がyyr2テーブルの開始を含んでいるか否かを判定する。それを含んでいる場合には、ブロック618からブロック620に移行して、入力ファイルからテーブル項目を読み出し、次いでブロック622で、yyr2テーブルに関してテーブル項目の調整を行う。それらの項目は、定義済の各パーサ状態毎に、そのパーサの状態についてユーザプロダクションコードを実行すべきか否かを示す情報を含めることにより、調整される。この調整に先立ち、yyr2アレイの下位ビットが、標準的なyaccパーサドライバに対し、そのパーサの特定の状態に関連した動作が存在するか否かを知らせる。また、その調整後には、第２パスに備えて、両方のパス中で実行すべきコードの有無を２パスパーサドライバに知らせるために、エクストラビットが付加される。このエクストラビットは、第１パスに関する第１ビットと、第２パスに関する第２ビットとを有している。テーブル項目の調整後に、ブロック622からブロック608に戻って、入力ファイルの次の行を読み出す。

【００４６】

また、前記行がyyr2テーブルの開始を含んでいない場合には、ブロック618からブロック702に移行して、その行が定数「YYACCEPT」に関するdefineステートメントを含んでいるか否かを判定し、それを含んでいる場合には、制御がブロック704に移行して、そのdefineステートメントを_YACCEPTに変更した後に図６のブロック608に戻って、次の入力行を読み出す。これは、元のソースコードとのネーミング上の矛盾を回避するために行われる。

【００４７】

また、前記行がYYACCEPTに関するdefineステートメントを含んでいない場合には、制御がブロック706に移行して、その行が「YYERROR」に関するdefineステートメントを含んでいるか否かを判定し、それを含んでいる場合には、ブロック706からブロック708に移行して、そのdefineステートメントを_YERRORに変更した後、図６のブロック608に戻る。これは、元のソースコードとのネーミング上の矛盾を回避するために行われる。

【００４８】

ブロック710では、入力行が行番号の指示語であるか否かを判定し、そうである場合には、ブロック712に移行して、行番号パラメータに３を加算してそれを出力へコピーした後、図６のブロック608に戻る。この行番号の調整は、修正済文法の始めに「#define TWOPASS」が付加されるからである。

【００４９】

これらが何れも生じない場合には、ブロック714で、出力ファイルに入力行を書き込み、次いでブロック716で、その行がcaseステートメントを含んでいるか否かを判定する。それが含まれている場合には、制御がブロック718に戻り、そのcaseステートメントのケース番号を変数LASTCASEに格納した後、ブロック720に移行する。ブロック720で、キャラクタストリング「/*__TWO_PASS:」により識別される２パスフラグが生じたか否かを判定し、それが生じていれば、ブロック722で、その２パスフラグからの最後のパス番号にＯＲ演算を行って上述の最後のケース番号に関するyyr2テーブル項目とする。ストリング「/*__TWO_PASS:」は、各パーサ動作毎にyyr2テーブルに対して如何なる調整を行うべきかについてポストプロセッサと連絡をとるためにプリプロセッサにより生成される。次いで制御は図６のブロック608に戻る。

【００５０】

ブロック610でファイルの終わりを検出すると、ブロック624に移行して、出力ファイルに新しいyyr2テーブルを書き込み、次いでブロック626で、ファイルをクローズした後、オペレーティングシステムに戻る。

【００５１】

図８および図９は、図５のブロック514から呼び出される関数yyparseを示すフローチャートである。同図では、図８のブロック802から入ることになる。ブロック802で、図１０のスキャナのトークン関数が呼び出される。スキャナは、入力ストリーム中で見つかった次のトークンを表す値を返す。ブロック802では、そのトークンがトークンPASS_DEFAULTであるか否か、即ち、そのトークンがトークンPASS_DEFAULTを表す値を有しているか否かを判定し、それがトークンPASS_DEFAULTである場合には、ブロック806に移行して、そのトークンに続く数を変数DEFAULTPASSに格納し、次いでブロック802に戻って、次のトークンを取得する。そのトークンが或る識別子を表している場合には、ブロック808からブロック810に移行して、その識別子をルックアップテーブルに格納し、次いでブロック814に移行して、そのトークンを出力ファイルに書き込んだ後、ブロック802に戻って次のトークンを取得する。次のトークンがトークンMARKでない場合には、ブロック812からブロック814に移行して、そのトークンを出力ファイルに書き込んだ後、ブロック802に戻って次のトークンを取得する。このようにして、第１マークが検出されるまで、入力ファイルが読み出される。

【００５２】

第１マークが検出された場合には、ブロック812からブロック816に移行して、そのマークを出力ファイルに書き込み、次いでブロック818で入力ファイルから次のトークンを取得する。ブロック820では、次のトークンがC_IDENTIFIERを表しているか否かを判定し、それを表している場合には、ブロック822に移行して、そのトークンをルックアップテーブルに格納し、次いでブロック826で、そのトークンを出力ファイルに書き込んだ後、ブロック818に戻って次のトークンを取得する。次のトークンがIDENFIERを表している場合には、ブロック824からブロック822に移行してその識別子をルックアップテーブルに格納し、次いでブロック826でそのトークンを出力に書き込んだ後、ブロック818に戻る。

【００５３】

また、次のトークンがC_IDENTIFIERもIDENTIFIERも表さない場合には、制御が図９のブロック902に移行して、そのトークンがトークンPASS_BLOCKであるか否かを判定する。そのトークンがトークンPASS_BLOCKでない場合には、ブロック902からブロック904に移行してそのトークンを出力に書き込んだ後、図８のブロック818に戻って次のトークンを取得する。また、前記トークンがトークンPASS_BLOCKである場合には、ブロック902からブロック906に移行してパス番号が現在既知のものであるか否かを判定し、既知のものである場合にはブロック914に移行する。そのパス番号が既知のものでない場合には、ブロック906からブロック908に移行してそのパス番号が確認されているか否かを判定し、そのパス番号が確認されている場合には、ブロック910に移行して、それまでに確認された最高のパス番号を使用し、次いでブロック914に移行する。パス番号が確認されなかった場合には、ブロック908からブロック912に移行して、それ以降のコードについてデフォルトパス番号を使用し、次いでブロック914に移行する。

【００５４】

ブロック914では、パス番号を示すコメントを出力ファイルに書き込み、次いでブロック916で、トークンが正確なパスでのみ実行されるようにするifステートメントを書き込んだ後、ブロック808に戻って次のトークンを取得する。前記ifステートメントは、それが実際に必要な場合にのみ書き込まれる。文法動作における全てのコードが２つのパスの一方でのみ実行される場合には、その各パスで実行されるコードをyyr2テーブルが制御することになる。また、パス１のコードとパス２のコードとが混在している場合にはifステートメントが必要となり、yyr2テーブルにより、文法動作を各パス毎に実行することが可能になる。ほとんどの場合、各文法動作は一方のパスでしか実行されないので、yyparse()でのスイッチステートメントに対する実際の呼び出しは不必要である。

【００５５】

この処理は、ブロック819でファイルの終わりを検出するまで続行され、その検出時に、図８から図５のブロック516に戻る。

【００５６】

図１０ないし図１２は、プリ／ポストプロセッサスキャナ404を示すフローチャートであり、図５ないし図７は、プリ／ポストプロセッサパーサ406を示すフローチャートである。

【００５７】

ここで、図１０ないし図１２を参照する。図８のブロック802およびブロック818からパーサによってスキャナが呼び出されると、制御が図１０のブロック1002に入る。ブロック1002で、入力ファイルから次のキャラクタを読み出し、ブロック1004で、キャラクタシーケンス「*{」または「1{」または「2{」または「12{」が生じたか否かを判定する。それが生じていない場合には、制御がブロック1006に移行して、シーケンス「pass1{」または「pass2{」または「pass12{」または「pass*{」が生じたか否かを判定する。ブロック1004またはブロック1006で識別されるそれらのシーケンスのうちの何れかが生じた場合には、制御がブロック1016に移行して、トークンPASS_BLOCKによって表される値に変数トークンの値をセットし、次いで制御が図１２のブロック1230に移行する。図１２のブロック1230では、その見つけたばかりのトークンのための構造を構築し、次いでブロック1232に移行して、そのトークンのための定数値（例えばPASS_BLOCKのための値）と前記の構成したばかりの構造とを呼び出し側のプログラムに返す。

【００５８】

また、ブロック1004,1006で識別されるシーケンスが生じなかった場合には、制御がブロック1008に移行して、入力ストリームから数が読み出されたか否かを判定する。数が読み出された場合には、制御がブロック1018に移行して、変数トークンをトークンNUMBERの値にセットし、制御が図１２のブロック1230に移行する。また、そのシーケンスが数でない場合には、制御がブロック1010に移行して、そのシーケンスが、記号「:」が後に続く有効な名称であるか否かを判定する。それが有効な名称である場合には、ブロック1020で、トークンをC_IDENTIFIERにセットして、制御が図１２のブロック1230に移行する。また、

コロンが後に続く名称が生じなかった場合には、制御がブロック1011に移行して、そのシーケンスが、コロンが後に続かない名称であるか否かを判定する。前記シーケンスが、コロンが後に続かない名称である場合には、制御がブロック1021に移行して、トークンの値をIDENTIFIERにセットし、次いで制御が図１２のブロック1230に移行する。ブロック1012,1014では、キャラクタ「%LEFT」および「%RIGHT」およびそれらの何れかが生じたか否かをチェックし、そのトークンの値をセットして、制御が図１２のブロック1230に移行する。

【００５９】

それらのシーケンスの何れも生じなかった場合には、制御が図１１のブロック1102に移行し、ブロック1102～1124で、パーセント符号で始まるキャラクタシーケンスを全てチェックし、それが見つかった場合には、トークンの値をセットして、制御が図１２のブロック1230に移行する。また、パーセント符号で始まるキャラクタシーケンスが全く見つからなかった場合には、制御がブロック1126に移行して、２つのパーセント符号（即ち、「%%」）からなるシーケンスが生じたか否かを判定する。このシーケンスが生じた場合には、制御がブロック1128に移行して、先行する等しいシーケンスが見つかったか否かを判定する。このシーケンスは、マークと呼ばれ、読み出されるソースコード内の主な区切りを識別する。先行するマークが見つかった場合、制御がブロック1130に移行して、入力ファイルの残りをキャラクタバッファにコピーし、次いでブロック1132に移行してトークンの値をMARKにセットし、図１２のブロック1230に移行する。先行するマークが見つからなかった場合には、バッファへのコピーは行わないが、トークンの値をセットして、制御がやはり図１２のブロック1230に移行する。マークシーケンスが見つからなかった場合には、制御は図１２のブロック1202に移行する。

【００６０】

図１２のブロック1202で、特殊記号のチェックを行い、何れかの特殊記号が見つかった場合には、ブロック1204で、変数トークンを実際のキャラクタにセットした後、ブロック1230に移行して、旧値の構造を構築して返す。また、読み出されたキャラクタが特殊記号のうちの１つでない場合には、制御がブロック1206に移行して、シーケンス「%TWOPASS_DEFAULT」を探索し、そのシーケンスが見つかった場合には、制御がブロック1208に移行してトークンをPASS_DEFAULTにセットして、ブロック1230に移行する。また、シーケンス「%TWOPASS_DEFAULT」が見つからなかった場合には、制御がブロック1210に移行して、シーケンス「%{」を探索し、それが見つかった場合には、ブロック1210からブロック1212に移行して、終了シーケンス「%}」が見つかるまで全ての入力キャラクタをキャラクタバッファにコピーする。それら２つのシーケンスの間に、本システムのユーザは、所望のあらゆるタイプの言語によるソースコードを挿入することが可能であり、そのソースコードは出力に直接コピーされることになる。その情報をキャラクタバッファにコピーした後、制御が図１０のブロック1002に戻って、次の入力文字の読み出しおよびその処理が行われる。

【００６１】

また、シーケンス「%{」が見つからなかった場合には、制御がブロック1214に移行して「#」を探索し、それが見つかった場合には、ブロック1216で、その行と、その第１行の終わりにある継続キャラクタに続く後続行とを、キャラクタバッファにコピーする。このようにして、ユーザは、言語マクロを入力し、出力に直接コピーすることが可能になる。このキャラクタバッファへのコピーを行った後、ブロック1216からブロック1002に戻って入力の次のキャラクタを読み出す。

【００６２】

また、符号「#」が見つからなかった場合には、ブロック1214からブロック1218に移行して、引用符中のキャラクタストリングが見つかったか否かを判定し、それが見つかった場合には、ブロック1220で、そのキャラクタストリングをキャラクタバッファへコピーして、制御がブロック1002に戻る。

【００６３】

また、引用符中のキャラクタストリングが見つからなかった場合には、ブロック1218から

ブロック1222に移行して、数字ストリングが後続するドル符号が見つかったか否かを判定する。それが見つからなかった場合には、制御がブロック1224に移行し、如何なるキャラクタが見つかろうともそれをキャラクタバッファにコピーし、次いで制御がブロック1002に戻って次のキャラクタを読み出す。また、数字が後続するドル符号が見つかった場合には、制御がブロック1226に移行して、その数を整数に変換し、ブロック1227で、変数トークンの値を定数DOLLARにセットし、次いで制御がブロック1230に移行して、そのトークンを呼び出し側のプログラムに返す。

【００６４】

図１３は、２パスコンパイラの動作を示すブロック図である。同図において、２パスコンパイラ126には、その２パスコンパイラのユーザにより供給されたソースコード1302が入力される。２パスコンパイラは、ソースコードファイル1302の処理を行ってパス１中に構文解析ファイル1306を生成する。この構文解析ファイル1306が、以下で説明するようにパス２中に読み返されて、出力1308が生成される。

【００６５】

図１４ないし図１６は、図２の２パスコンパイラ126のためのパーサを示すフローチャートである。この２パスコンパイラ126のパーサセクションは、２パスパーサドライバ210からコピーされ、ファイル208を介してパーサジェネレータ122から受け取ったユーザプロダクションコードと共に組み込まれる。ポストプロセッサ124は、ファイル208からのユーザプロダクションコードを２パスパーサドライバ210からのコードと組み合わせてソースファイル212を生成し、そのソースファイル212が、コンパイラ214によってコンパイルされた後に、２パスコンパイラ126になる。もちろん、ユーザプロダクションコードは、言語文法ファイル204と共にユーザにより供給されるものであり、これ以上の説明は省略することとする。

【００６６】

次に、図１４ないし図１６を参照する。２パスコンパイラが処理を開始すると、制御が図１４のブロック1402に入る。ブロック1402では、パーサで使用する全ての変数に初期値をセットし、ブロック1404で図２１に示す関数を呼び出して、パーサを書込モードに切り換える。パーサが書込モードにある場合には、パーサの各状態が、コンパイラのパス２での使用に備えて構文解析ファイルに書き込まれる。

【００６７】

構文解析ファイルは、下記の形態の２バイト制御レコードからなる線形シーケンスである。

【００６８】

ビット0～10 ：新しい状態、または、特殊レコードを示す特殊トークン値。

【００６９】

ビット11 ：多用途：これは、通常のパーサ状態に関してデータレコードが後続する(twopass_saveから)ことを示す。また、前記状態がこれを特殊レコードtwopass_callfuncと識別する場合にはエラー関数を示す。また、デルタTWOPASS_VARの上位ビットを提供する。

【００７０】

ビット12～15：新しい状態が特殊レコードを示す場合には、特殊コードのタイプである。通常の状態の場合には、yaccに加えるべき新しいトークンの数がスタックされる。

【００７１】

制御レコードに加えて、その制御レコードにデータレコードが後続することができる。

【００７２】

書込モードに切り換えた後、制御がブロック1406に戻って、ユーザにより供給された関数twopass_begin_pass1を呼び出す。言語文法204（図２参照）内のユーザコードにより供給されるその関数によって、ユーザが、パス１コードの開始に先立ってあらゆる所望の操作を行うことが可能になる。この関数は、必要に応じて設けられるものであり、ユーザが組み込まねばならないものではない。

【００７３】
ユーザにより供給された関数から戻った後、ブロック1408で、処理中のソースファイルから次のトークンを取得する。そのトークンは、スキャナ関数を利用して得られる。このスキャナは、ユーザにより供給され、図１０ないし図１２に関連して上述したスキャナと同様の方法で、字句プログラムを用いて定義することが可能なものである。次いでブロック1410で、ソースファイルの終わりに達したか否かを判定し、達していない場合には、制御がブロック1412に移行して、受信したトークンの構文解析を行って、実行すべきプロダクションコードを（もし存在すれば）判定する。この構文解析は、上述の当業界で周知のyaccパーサジェネレータを用いて行われる。構文解析の実行プロセスでは、パーサは、特定のパーサ状態に達してその状態をブロック1414に渡し、同ブロックで前記状態が構文解析ファイル(PARSEFILE)に書き込まれ、それがパス２で利用されることになる。構文解析ファイルへの状態の書き込みが終了した後、ブロック1416で、その状態に関するユーザ動作の有無を判定する。これは、動作を定義する、yyr2テーブルへの索引として、前記状態を用いることにより判定される。その状態に関するyyr2テーブル中の項目が、動作の存在を示している場合には、ブロック1416からブロック1418に移行してパラメータ値をセットする。次いでブロック1420で、その状態に関して定義されたユーザパス１またはユーザパス12動作を実行する。従って、パス２またはパス12として指定された全ての動作が、この処理ポイントで実行されることになる。ユーザ動作の実行後、ブロック1420からブロック1408に戻って、次のトークンを得る。また、ユーザ動作が存在しない場合には、ブロック1416からブロック1422に移行してyaccのスタックを一掃した後、ブロック1408に戻る。このループは、関数twopass_dopass2（以下の図２８参照）に遭遇しない限り、入力ソースファイルの終わりに達するまで続行される。前記関数twopass_dopass2に遭遇した際には、このループは、パス２の処理が完了するまで中断され、その後に再開されることになる。
【００７４】
従って、パス１の処理は、各パーサ状態が構文解析ファイルにセーブされ、および、パス１またはパス12中に実行されるように定義されたプロダクションだけが実際に実行されることになるという点を除けば、従来の周知のコンパイラと同様のものである。入力ソースファイルの終わりに達すると、ブロック1410から図１５のブロック1502に移行する。
【００７５】
図１５および１６は、２パスコンパイラの第２パスに関する処理を示すフローチャートである。ブロック1502で、ユーザにより供給される関数twopass_begin_pass2を呼び出す。上述の関数twopass_begin_pass1と同様に、ユーザは、パス２処理の開始に先立って実行されることになるコードを供給することができる。ユーザ関数から戻った後、ブロック1504で図２２の関数を呼び出して、構文解析ファイルの処理を読出モードに切り換える。次いでブロック1506で、構文解析ファイルから次の状態を読み出し、ブロック1507で、ファイルの終わりに達したか否かを判定する。ファイルの終わりに遭遇した場合には、制御が呼び出し側のプログラムに戻り、それ以外の場合には、ブロック1508で、その状態が特殊状態であるか否かを判定する。特殊状態については以下で定義することとする。その状態が特殊状態でない場合には、ブロック1508からブロック1510に移行して、その状態をyaccスタックに加え、その処理が可能になるようにする。
【００７６】
実際には、２つのyaccスタック、即ち、処理を受ける状態のためのスタック（以下、状態スタックと称す）と、各プロダクションに関連するトークン（および先行プロダクションからの戻り値）のためのスタック（以下、トークンスタックと称す）とが存在する。第１パス中には、状態スタックにプッシュされる状態が、トークンスタックにおける最近の変更（最後の削減以降に幾つのトークンが加えられたか）と共に中間ファイルに格納される。また、第２パス中では、状態スタックは問題とされない。これは、その状態が中間ファイル中で線形順になるからである。しかし、状態情報と共に保持されたトークンのカウントが与えられた場合にはトークンスタックは維持される。従って、第２パスでは状態スタックは修正されず、トークンスタックのみが加えられる。また、両方のパスで、yyr2テー

ブル中の除去すべきトークンのカウントが与えられた場合には、トークンスタックは、プロダクションの削減によって削減される。

【００７７】

次いでブロック1512で、その状態に関するユーザ動作の有無を判定する。これは、動作を定義する、yyr2テーブルへの索引として、前記状態を利用することにより判定される。この状態に関するyyr2テーブル中の項目が動作の存在を示している場合には、ブロック1512からブロック1514に移行して、ブロック1506で読み出したデータを取得し、パス１処理におけるそのポイントで存在したように、パラメータのパラメータ値をセットする。次いでブロック1518で、その状態について定義されたユーザパス１またはユーザパス12動作を実行する。従って、パス２またはパス12として指定された全ての動作は、この処理ポイントで実行されることになる。ユーザ動作の実施後、ブロック1518からブロック1506に戻って、次の状態を読み出す。

【００７８】

ブロック1512で、この状態に関するユーザ動作が存在しないことを判定した場合には、ブロック1516に移行して、構文解析ファイルからの次の状態の読み出しに備えてyaccスタックを一掃した後、ブロック1506に戻って次の状態を取得する。従って、上述のようにして、元の入力ソースファイルからのトークンの再構文解析を必要とすることなく、２パス関数が全て達成される。

【００７９】

また、ブロック1508で、前記状態が特殊状態であると判定した場合には、ブロック1508から図１６のブロック1602に移行する。特殊状態とは、構文解析ファイルに挿入されてパス２の処理のための条件を設定する状態である。

【００８０】

ブロック1602では、その状態が状態END_OF_DOPASS2状態であるか否かを判定する。本システムのユーザが要求することのできる操作の１つに、パス１処理の中断およびパス２の即時処理がある。これは、ユーザプロダクションコード内のマクロTWOPASS_DOPASS2を用いて行われる。そのマクロに遭遇した場合には、図２８のフローチャートが処理され、その結果としてパス１処理からパス２処理に切り換わる。この切り換えが生じる前に、トークンEND_OF_DOPASS2が構文解析ファイル中に格納される。パス２処理中にこのトークンに遭遇した場合には、ブロック1602からブロック1604に移行する。次いでブロック1604で、ユーザにより供給された関数twopass_begin_pass1の２度目の実行を行い、または、マクロが多数回にわたって用いられる場合には後続回数だけ実行し、次いでブロック1606で図２１の関数を呼び出して書き込み状態へと逆に切り換わる。図２１でその処理が完了した場合には、制御が図１４のブロック1408に戻って、パス１処理が再開される。

【００８１】

前記特殊状態が状態END_OF_DOPASS2でない場合には、ブロック1602からブロック1608に移行して、その状態が状態TWOPASS_CALLFUNCであるか否かを判定する。ユーザが２パス呼び出し関数操作を実行すると、図１９で説明するコードが実行されて、状態TWOPASS_CALLFUNC、および、パス２中で呼び出すべき関数の名称が、その関数を呼び出すためのパラメータと共に構文解析ファイルに格納される。この状態にパス２処理中に遭遇した場合には、ブロック1608からブロック1610に移行して図２５の関数を呼び出して、構文解析ファイルからパラメータを読み出す。次いでブロック1612で、それらのパラメータをメモリ中に復元し、ブロック1614で、元のtwopass_callfuncステートメントで指定された関数を呼び出す。その関数の実行後、制御は図１５のブロック1506に戻る。

【００８２】

その状態が、状態TWOPASS_CALLFUNCでない場合には、制御がブロック1616に戻って、その状態が変数復元状態であるか否かを判定する。それが変数復元状態である場合には、制御がブロック1618に移行して図２５を呼び出して、その変数に関するデータを読み出し、次いでブロック1620で、その読み取ったデータを変数に格納する。その変数の復元後、または、前記状態が変数復元状態でない場合には、制御が図１５のブロック1506に戻る。

【００８３】
図１７は、本システムのユーザが文法中の任意のポイントで呼び出して変数のセーブおよび復元を行うことが可能な関数TWOPASS_SAVEを示すフローチャートである。この関数はユーザプロダクションコードにより呼び出されるので、図１４ないし図１６の記述にはその呼び出しは直接示していない。この関数が呼び出されると、ブロック1702で、まず要求される変数のサイズを取得する。次いでブロック1704で、パス１が処理中か否かを判定し、パス１が処理中である場合には、ブロック1706に移行して、変数の内容をyysave_bufferに移送する。この内容は、上述のブロック1414で構文解析ファイルにパーサ状態が書き込まれる際に書き込まれることになる。また、パス１が処理中でない場合には、ブロック1704からブロック1708に移行して、パス２が処理中であるか否かを判定する。パス２が処理中である場合には、ブロック1708からブロック1710に移行して、構文解析ファイルから変数内容を読み出してメモリ中の変数位置に戻す。その内容のセーブ後にその内容を復元し、また、処理中のパスがパス１またはパス２でない場合には、図１７からその呼び出し側のプログラムに戻る。twopass_saveは、ソースファイル中の同じポイントにおいて第１パスまたは第２パスで呼び出される、という点に留意されたい。

【００８４】
図１８は、上述の関数twopass_restoreを示すフローチャートである。文法内または文法外からこの関数を呼び出すと、ブロック1802で、そのコードがパス１内にあるか否かを判定し、そのコードがパス１内にない場合には、単に呼び出し側のプログラムに戻ることになる。これは、パス２中での変数の復元が、図１６のパス２の特殊状態コードで行われ、詳細には、ブロック1618,1620で行われるからである。パス１が処理中である場合には、ブロック1802からブロック1804に移行して、構文解析ファイルに変数復元状態を書き込み、次いでブロック1806で、構文解析ファイルにデータタイプを書き込む。最後に、ブロック1808で、構文解析ファイルに実際のデータを書き込み、これが、上述のように復元されることになる。

【００８５】
twopass_saveおよびtwopass_restoreは、互いに関連のないものであり、独立した関数である。

【００８６】
twopass_saveは、どのパスが関連するかに基づいて２つの異なるものをセーブする。第１パス中には、twopass_saveは変数の内容を中間ファイルにセーブする。ファイルのサイズを縮小するために、何処でまたは如何に変数を取得したかに関するレコードがその変数と共にファイル中に格納されることはなく、そのデータのみが格納されることになる。また、第２パス中には、twopass_saveは、第１パス中で以前に格納された変数の値を復元する。第２パスでは、中間ファイル中に存在するものに関する情報が必要となるが、第１パス中における同じtwopass_saveの呼び出しが第２パス中で再び実行されるので、前記情報は暗黙的に提供され、従って、第２パスで実行される際には、第１パスで有していた同じ情報が全て暗黙的に備わっていることになる。

【００８７】
twopass_restoreは、文法外の変数に用いられる。twopass_restoreを第２パス中の場合と全く同様に再度実行する（これは文法外での実行が困難なものである）ことを要求する代わりに、第１のtwopass_restoreが、特殊レコードを中間ファイルに格納する。その特殊レコードは、データと、そのデータが復元されるべき位置との両方を含むものである。このレコードが第２パス中で処理される場合、そのデータは自動的に変数に戻される。

【００８８】
図１９は、本システムのユーザが利用可能な関数twopass_callfuncを示すフローチャートである。上述のように、ユーザがこの関数を利用する場合には、実際の関数呼び出しがパス１からパス２へ据え置かれる。ユーザは、パス２で呼び出すべき関数の名称と、その関数に渡すべき引数とを提供する。関数が実行されると、ブロック1902で図１９のフローチャートが呼び出される。ブロック1902では、ユーザにより指定された引数のリストを作成

し、次いでブロック1904で、その関数がパス１またはパス２から呼び出されているか否かを判定する。関数がパス２から呼び出されている場合には、ブロック1904からブロック1906に移行して、引数リストを用いて関数を呼び出し、次いで呼び出し側のプログラムに戻る。また、関数がパス１から呼び出されている場合には、ブロック1904からブロック1908に移行して図２０を呼び出して、上述のように全てのTWOPASS_VARのデータをセーブする。次いでブロック1910で、それがtwopass_callfuncであること、幾つの引数が含まれるか、および、その呼び出しが特殊エラータイプであるか否か、を示す特殊レコードを、構文解析ファイルに書き込む。次いでブロック1912で、引数のタイプを構文解析ファイルに書き込む。次いでブロック1914で、関数のポインタを構文解析ファイルに書き込み、次いでブロック1916で、引数を構文解析ファイルに書き込む。これらが全て読み出されて復元されて、パス２中で関数が呼び出されることになる。

【００８９】

図２０は、図１９の関数から呼び出される関数yy_syncvarを示すフローチャートである。この処理に入った後、ブロック2002で、デルタがセーブされているか否か、あるいは、全変数がセーブされているか否かを判定する。デルタは、変化が小さい場合に利用される。これは、小さな数づつインクリメントするのが普通である行番号等のデータにとって極めて有用なものである。構文解析ファイル中のレコードは、小さなデルタが３ビット（最大値８まで）に納まるような態様で構成される。変化がそのスペースに納まらない場合には、新しい値が書き込まれる。デルタは、中間ファイルのサイズを縮小するために用いられるが、絶対不可欠なものではない。

【００９０】

デルタがセーブされている場合には、ブロック2002からブロック2004に移行して、何れかのデータが変更されたか否かを判定し、変更がない場合には、単純に呼び出し側のプログラムに戻る。また、データに変更がある場合、またはデルタがセーブされていない場合には、制御がブロック2006に移行して、TWOPASS_VARのデータを構文解析ファイルに書き込む。そのデータは、図１６に関連して上述した特殊状態のパス２処理中に復元されることになる。

【００９１】

図２１は、図１４および図１６から呼び出される書込モード切換関数を示すフローチャートである。同図の処理に入った後、ブロック2102で、メモリ中のバイト書込カウンタを調べることにより、構文解析ファイルに何か書き込まれたか否かを判定する。構文解析ファイル中に何らかのデータが書き込まれている場合には、ブロック2102からブロック2104に移行して、構文解析ファイルを巻戻し(rewind)、次いでブロック2106で、バッファ書込フラグをクリアする。そのフラグのクリア後、または、構文解析ファイルに何も書き込まれていない場合には、制御がブロック2108に移行して、構文解析バッファのポインタを始端にリセットし、次いでブロック2110で、変数TWOPASS_VARの全てをセーブ領域にコピーした後、呼び出し側のプログラムに戻る。

【００９２】

構文解析ファイル中では、不必要なI/0を回避するためにデータはできる限り圧縮される。また、大きなバッファを用いて不必要なI/0が回避される。多くの場合、中間ファイルは、バッファサイズを超えて成長することはないので、周辺装置に対するI/0の必要が回避される。また、構文解析ファイルは、逐次の読み出しおよび書き込みしか行われず、速度を高めるために探索は行われない。

【００９３】

図２２は、図１５から呼び出される読出モード切換関数を示すフローチャートである。同図の処理に入った後、ブロック2202で、メモリ中のバイト書込カウンタを調べることにより、構文解析ファイルに何か書き込まれたか否かを判定する。構文解析ファイルに何か書き込まれている場合には、ブロック2202からブロック2206に移行して、構文解析バッファの残りのデータを構文解析ファイルに書き込み、次いでブロック2208で構文解析ファイルを巻戻す。次いでブロック2210で、構文解析バッファが保持することになるのと同量のデ

ータを構文解析ファイルから読み出す。

【００９４】

また、構文解析ファイルに何も書き込まれていない場合には、ブロック2202からブロック2204に移行して、構文解析バッファのポインタをリセットし、そのバッファの始端を指すようにする。そのポインタのリセット後、またはブロック2210で構文解析バッファへのデータを読み出した後、制御がブロック2212に移行して、上述の書込モード切換関数により変数TWOPASS_VARのデータが格納されたセーブ領域からそのデータの全てをコピーして、呼び出し側プログラムに戻る。

【００９５】

図２３は、図１４から呼び出される状態書込関数を示すフローチャートである。同図において、ブロック2301で、構文解析バッファ中のデータの有無を示すフラグを構造yysave中にセットする。次いでブロック2302で、その構造yysaveを構文解析ファイルに書き込み、ブロック2303で、バッファ中のデータの有無を判定し、バッファ中にデータが存在する場合には、ブロック2304に移行して、そのバッファ中のデータを構文解析ファイルに書き込む。

【００９６】

図２４は、上述の多数のフローチャートから呼び出される関数yywriteを示すフローチャートである。同図の処理に入った後、ブロック2402で、構文解析バッファにパラメータをコピーし、次いでブロック2404で、構文解析バッファが満杯か否かを判定する。構文解析バッファが満杯の場合には、ブロック2404からブロック2406に移行して、その構文解析バッファの内容を構文解析ファイルに書き込み、次いでブロック2408で、構文解析ファイルへのデータ書き込みがあったことを示す書込バイトカウントをバイト書込カウンタに加える。構文解析ファイルへのデータ書き込み後、または、バッファが満杯ではない場合には、図２４からその呼び出し側のプログラムに戻る。

【００９７】

図２５は、上述のフローチャートの様々な位置から呼び出される関数yyreadを示すフローチャートである。同図の処理に入った後、ブロック2502で、読み出すよう要求されたデータを構文解析バッファからパラメータへコピーする。次いでブロック2504で、構文解析バッファ中にデータが残っているか否かを判定し、データが残っていない場合には、ブロック2506に移行して、構文解析ファイルからバッファデータを読み出して構文解析バッファの再充填を行う。構文解析バッファの再充填後、または、構文解析バッファ中に更にデータが含まれている場合には、図２５から呼び出し側プログラムに戻る。

【００９８】

図２６は、２パスコンパイラで利用可能な関数TWOPASS_SAVE_LOCを示すフローチャートである。関数TWOPASS_STUFF_LOCと連係して利用されるこの関数により、ユーザは、情報をセーブし、その情報がセーブされた位置よりも早期の処理位置においてパス２中でその情報を利用できるようにすることが可能になる。関数TWOPASS_STUFF_LOCは呼び出す必要はないが、その場合には、関数TWOPASS_SAVE_LOCは関数twopass_saveと同様の働きをする。

【００９９】

ここで図２６を参照する。同図の処理に入った後、ブロック2602で、次の変数が格納されることになる構文解析バッファ中の位置を判定する。ブロック2604で、その位置を構文解析バッファに書き込み、次いでブロック2606で図１７（twopass_save）を呼び出して変数をセーブする。次いでブロック2608で、その位置をユーザに返す。

【０１００】

図２７は、関数TWOPASS_STUFF_LOCを示すフローチャートである。同図の処理に入った後、ブロック2702で、早期のTWOPASS_SAVE_LOC呼び出しからの位置を用いて構文解析ファイルから状態を読み出す。次いでブロック2704で、その状態がSAVE_LOCであるか否かを判定し、その状態がSAVE_LOCである場合には、ブロック2706に移行して、関数TWOPASS_SAVE_LOCにより識別された位置に新しいデータをセーブする。そのデータのセーブ後、または、前記状態がSAVE_LOCでない場合には、制御が呼び出しプログラムに戻る。

【０１０１】
図２８は、ユーザが利用可能な関数twopass_dopass2を示すフローチャートである。この関数により、２パスコンパイラは、パス１の操作を中断し、ソースファイルに関して既に定義済みの全てのパス２操作を実行することになる。同図の処理に入った後、ブロック2802で、構文解析バッファを構文解析ファイルに書き込む。次いでブロック2804で、システムスタックに現在のパーサ状態をセーブして、パス２処理の完了後にパス１処理を再開できるようにする。次いでブロック2806で、特殊状態END_OF_DOPASS2を構文解析ファイルに書き込み、図１５に移行してパス２処理を開始する。図１５が特殊状態END_OF_DOPASS2に遭遇すると、パス１処理が再開される。
【０１０２】
以下に示すリストは、本発明の２パスコンパイラを利用して実施されたコンパイラの一例を示すものである。リスト１（表１ないし表５）は入力ソースを示し、リスト２（表６ないし表１０）はそのソースに関する中間ファイル410（図４参照）を示している。また、参考資料Ａは、本システムに関するユーザーズガイドを含み、参考資料Ｂは、図１０ないし図１２で説明した字句スキャナのためのソースを含み、参考資料Ｃは、図５ないし図９で説明したパーサについてのソースコードを含み、参考資料Ｄは、図１３ないし図２８で説明した実行時間パーサについてのソースコードを含むものである。
【０１０３】
【表１】

リスト１

```
%{ /* -*-C-*- */

/*  方程式解答プログラム（非常に単純な形の方程式）；２パスプログ
 *  ラムのサンプル
 *
 *  このプログラムは、次のような形のいくつかの方程式で構成される
 *  ファイルを入力とする。
 *      v  =  expr
 *  ここで、vは１文字の変数名で、exprは定数と変数からなる、
 *  中置表記法で表される式である。
 *      y  =  constant
 *  の様な形の式は定数式とみなされ、この場合yは定数変数である。
 *  このプログラムの目的のため、定数のみからなる式でも非定数式と
 *  みなされる。任意の負符号を持った単精度浮動小数点数のみが定数
 *  とみなされる。
 *
 *  このプログラムは非定数変数全ての値を計算し、入力ファイルを読
 *  み終わった後に結果を印刷出力する。
 *
 *  入力ファイル中の全ての空白文字は無視されるが、復帰改行文字は
 *  式の終わりを示す。
 *
 *  このプログラムが１パスプログラムであったなら、入力ファイルに
 *  は前もって定義された位置にしか定数変数を使うことができない。
 *  ２パスプログラムであれば、入力ファイルのどの位置にあっても
 *  定数変数を使うことができる。
 *  他の非定数変数もまた、それが使われる前に位置が定義され
 *  使用される。
 *
 *  入力ファイル例
 *    r  =  5
 *    a  =  r*r*p
 *    s  =  2*r*p
 *    t  =  a/r*2
 *    p  =  3.1415
 *  出力結果
 *    a  =  78.537498
 *    p  =  3.141500
 *    r  =  5.000000
 *    s  =  31.415001          <---- 典型的な浮動小数点
 *    t  =  31.414999          <--/  計算における誤差
 *
 *  'p'（パイの頭文字を意味する）は、ファイル内のどの位置に
 *  あるか定義されることなく使用されていることに注意する。
 *  しかし、定数変数なので使用可能である。第１パスはこれらの
 *  定数を第２パスで使用する為に取り出す。第２パスで'a'の
 *  値を定義した為に't'の式がそれを使うことができるように
 *  なったことも重要なことである。
 *
```

【０１０４】

【表２】

```
 *  これがマルチパスプログラムならば、定数と非定数変数の区別が
 *  不必要となり、矛盾する式を発見するかまたは、与えられた式の
 *  最後になるかまで、方程式の処理を繰り返すことができる。
 */

#include <stdio.h>
#include <ctype.h>

struct
{
    int full;          /* 変数が値を持つことを示すフラグ */
    float val;         /* 変数に与えられた値 */
} variable['z'+1];

int vals;              /* 1 = 定数式、   >1 = 非定数式 */
%}

%union{
        float num;     /* 浮動小数点数（ユーザが与えた定数）*/
        int var;       /* 変数（変数名を表すアスキーコード）*/
}

%token NUM VAR

%type <num> expr NUM val
%type <var> VAR

%left '+' '-'
%left '*' '/'
%right UMINUS

%start set

%%

set:    equ
        2{ int i;
            for( i = 'a'; i <= 'z'; i++ )
                if( variable[i].full )
                {
                  printf( "%c = %f\n", i, variable[i].val );
                  variable[i].full = 0;
                }
         }
        ;

equ:     assign equ
       | /* 処理なし */
       ;
```

【０１０５】
【表３】

```
assign:   VAR '=' 1{ vals = 0; } expr '\n'
        *{
           1{ if( vals == 1)
              {
                    variable[$1].val = $4;
                    variable[$1].full = 1;
              }
           }
           TWOPASS_SAVE($1);
           2{
              variable[$1].val = $4;
              variable[$1].full = 1;
            }
         }
         ;

val:       NUM
         *{
           TWOPASS_SAVE( $1 );
           1{ vals++;  }
           2{ $$ = $1; }     /*  $1 を変更した場合は、
                              *  $$ にも変更が必要
                              */
         }
       |   VAR
         *{
           TWOPASS_SAVE( $1 );
           1{ vals += 2; }
           2{
            if( variable[$1].full )
              $$ = variable[$1].val;
             else
              fprintf( stderr,
                      "Use of variable '%c' before definition",
                      $1 );
         }  }
        ;
/* UMINUSは負の定数を扱うために第1、第2両方のパスで
 * 処理されることが必要
 */
expr:      expr '+' expr              2{ $$ = $1 + $3; }
         | expr '-' expr              2{ $$ = $1 - $3; }
         | expr '*' expr              2{ $$ = $1 * $3; }
         | expr '/' expr              2{ $$ = $1 / $3; }
         | '(' expr ')'               2{ $$ = $2; }
         | '-' expr    %prec UMINUS   *{ $$ = - $2; }
         | val
         ;

%%
```

【０１０６】
【表４】

```
#ifdef YYDEBUG
int yylex()  { int c = real_yylex();
               printf( ">> Token: '%c' (%d)\n", c, c);
               return( c ); }
int real_yylex()
#else
int yylex()
#endif
{
    static int equseen = 0;
    int c;

    while( 1 )
    {
        switch( c = getchar() )
        {
         case EOF:
           return( 0 );
           break;
         case '0': case '1': case '2': case '3': case '4':
         case '5': case '6': case '7': case '8': case '9':
         case '.':
            ungetc( c, stdin );
            scanf( "%f", &yylval.num );
            return( NUM );
            break;
         case '=':
           equseen = 1;              /* (ブレークせずに)
                                      *  そのまま下へ
                                      */
         case '-': case '+': case '*': case '/':
         case '(': case ')':
           return( c );
           break;
         case '\n':
           if( equseen)  { equseen = 0; return( '\n' ); }
           break;
         case ' ': case '\t':      /* 空白文字は無視される */
           break;
         default:
           if( isalpha( c ) )
           {
               yylval.var = (isupper(c) ? tolower(c) : c);
               return( VAR );
            }
            break;
         }
      }
}
```

【０１０７】
【表５】

```
main()
{
#ifdef YYDEBUG
    extern int yydebug;
    yydebug = 1;
#endif
    yyparse();
}

/* これらのコードは今回不必要 */
twopass__begin__pass1()  {}
twopass__begin__pass2()  {}
yyerror()                {}
```

【０１０８】
【表６】

リスト２

```
%{
#define TWOPASS 1
%}
%{ /* -*-C-*- */

/*  方程式解答プログラム（非常に単純な形の方程式）；２パスプログ
 *  ラムのサンプル
 *
 *  このプログラムは、次のような形のいくつかの方程式で構成される
 *  ファイルを入力とする。
 *      v  =  expr
 *  ここで、vは1文字の変数名で、exprは定数と変数からなる、
 *  中置表記法で表される式である。
 *      y  =  constant
 *  の様な形の式は定数式とみなされ、この場合yは定数変数である。
 *  このプログラムの目的のため、定数のみからなる式でも非定数式と
 *  みなされる。任意の負符号を持った単精度浮動小数点数のみが定数
 *  とみなされる。
 *
 *  このプログラムは非定数変数全ての値を計算し、入力ファイルを読
 *  み終わった後に結果を印刷出力する。
 *
 *  入力ファイル中の全ての空白文字は無視されるが、復帰改行文字は
 *  式の終わりを示す。
 *
 *  このプログラムが1パスプログラムであったなら、入力ファイルに
 *  は前もって定義された位置にしか定数変数を使うことができない。
 *  ２パスプログラムであれば、入力ファイルのどの位置にあっても
 *  定数変数を使うことができる。
 *  他の非定数変数もまた、それが使われる前に位置が定義され
 *  使用される。
 *
 *  入力ファイル例
 *    r  =  5
 *    a  =  r*r*p
 *    s  =  2*r*p
 *    t  =  a/r*2
 *    p  =  3.1415
 *  出力結果
 *    a  =  78.537498
 *    p  =  3.141500
 *    r  =  5.000000
 *    s  =  31.415001          <---- 典型的な浮動小数点
 *    t  =  31.414999          <--/  計算における誤差
 *
 *  'p'（パイの頭文字を意味する）は、ファイル内のどの位置に
 *  あるか定義されることなく使用されていることに注意する。
```

【０１０９】
【表７】

```
 *  しかし、定数変数なので使用可能である。第１パスはこれらの
 *  定数を第２パスで使用する為に取り出す。第２パスで’ａ’の
 *  値を定義した為に’ｔ’の式がそれを使うことができるように
 *  なったことも重要なことである。
 *
 *  これがマルチパスプログラムならば、定数と非定数変数の区別が
 *  不必要となり、矛盾する式を発見するかまたは、与えられた式の
 *  最後になるかまで、方程式の処理を繰り返すことができる。
 */

#include <stdio.h>
#include <ctype.h>

struct
{
    int full;          /* 変数が値を持つことを示すフラグ */
    float val;         /* 変数に与えられた値 */
} variable['z'+1];

int vals;               /* 1 = 定数式、   >1 = 非定数式 */
%}

%union {
        float num;      /* 浮動小数点数（ユーザが与えた定数）*/
        int var;        /* 変数（変数名を表すアスキーコード）*/
}

%token NUM VAR

%type <num> expr NUM val
%type <var> VAR

%left '+' '-'
%left '*' '/'
%right UMINUS

%start set

%%

set:   equ
       { pass2{ /*__TWOPASS__PASS:2*/  int i;
          for( i= 'a'; i <= 'z'; i++ )
               if( variable[i].full )
               {
                 printf( "%c = %f\n", i, variable[i].val );
                 variable[i].full = 0;
               }
        }}
      ;
```

【０１１０】

【表８】

```
equ:    assign equ
        | /* 処理なし */
        ;

assign:    VAR '='
           { pass1{ /*__TWOPASS__PASS:1*/ vals = 0; }} expr '\n'

        { pass12{ /*__TWOPASS__PASS:3*/
           { if(yydopass&8) pass1{ if( vals == 1 )
              {
                    variable[$1].val = $4;
                    variable[$1].full = 1;
              }
           }}
           TWOPASS__SAVE( $1 );
           { if(yydopass&2) pass2{
             variable[$1].val = $4;
             variable[$1].full = 1;
           }}
        }}
        ;

val:       NUM
        { pass12{ /*__TWOPASS__PASS:3*/
           TWOPASS__SAVE( $1 );
           { if(yydopass&8) pass1{ vals++; }}
           { if(yydopass&2) pass2{ $$ = $1; }}
                /* $1 を変更した場合は、$$ にも変更が必要*/
     }}
     |    VAR
          { pass12{ /*__TWOPASS__PASS:3*/
           TWOPASS__SAVE( $1 );
           { if(yydopass&8) pass1{ vals += 2; }}
           { if(yydopass&2) pass2{
               if( variable[$1].full )
                 $$ = variable[$1].val;
               else
                 fprintf( stderr,
                        "Use of variable '%c' before definition",
                         $1 );
       }}  }}
     ;

/* ＵＭＩＮＵＳは負の定数を扱うために第１、第２両方のパスで
 * 処理されることが必要
 */
```

【０１１１】

【表９】

```
expr:   expr '+' expr {pass2(/*__TWOPASS__PASS:2*/$$= $1 + $3;}}
      | expr '-' expr {pass2(/*__TWOPASS__PASS:2*/$$= $1 - $3;}}
      | expr '*' expr {pass2(/*__TWOPASS__PASS:2*/$$= $1 * $3;}}
      | expr '/' expr {pass2(/*__TWOPASS__PASS:2*/$$= $1 / $3;}}
      | '(' expr ')'  {pass2(/*__TWOPASS__PASS:2*/$$= $2; }}
      | '-' expr  %prec UMINUS
                      {pass12(/*__TWOPASS__PASS:3*/$$= - $2; }}
      | val
      ;
%%

#ifdef YYDEBUG
int yylex() { int c = real__yylex();
              printf( ">> Token: '%c' (%d)\n", c, c);
              return( c ); }
int real__yylex()
#else
int yylex()
#endif
{
    static int equseen = 0;
    int c;

    while( 1 )
    {
        switch( c = getchar() )
        {
         case EOF:
           return( 0 );
           break;
         case '0': case '1': case '2': case '3': case '4':
         case '5': case '6': case '7': case '8': case '9':
         case '.':
            ungetc( c, stdin );
            scanf( "%f", &yylval.num );
            return( NUM );
            break;
         case '=':
           equseen = 1;              /* (ブレークせずに)
                                      *  そのまま下へ
                                      */
         case '-': case '+': case '*': case '/':
         case '(': case ')':
           return( c );
           break;
         case '\n':
           if( equseen)  { equseen = 0; return( '\n' ); }
           break;
```

【０１１２】
【表１０】

```
            case ' ': case '\t':      /* 空白文字は無視される */
              break;
            default:
              if( isalpha( c ) )
              {
                  yylval.var = (isupper(c) ? tolower(c) : c);
                  return( VAR );
              }
               break;
             }
        }
}

main()
{
#ifdef YYDEBUG
     extern int yydebug;
     yydebug = 1;
#endif
     yyparse();
}
/* これらのコードは今回不必要 */
twopass__begin__pass1()  {}
twopass__begin__pass2()  {}
yyerror()                {}
```

【０１１３】
以上で、本発明に関する現時点での好適実施例についての説明を終えるが、本発明の態様が完全に実施されたことが理解されよう。また、当業者には明らかなように、本発明の思想および範囲を逸脱することのない本発明の構成および回路に対する多数の変更や多種多様な実施例および用途は自明のものである。本明細書における開示および説明は、例示を目的としたものであり、本発明の制限を意図したものではなく、本発明の範囲は特許請求の範囲で好適に規定されている。

【０１１４】
以下においては本発明の種々の構成要件の組み合わせからなる例示的な実施態様を示す。

【０１１５】
１．第１のコンピュータ言語を第２のコンピュータ言語にコンパイルするためのコンピュータ実施方法であって、前記コンパイルが少なくとも２回のコンパイルパスを必要とするものにおいて、前記コンピュータ実施方法が、
前記第１のコンピュータ言語を前記第２のコンピュータ言語にコンパイルするためのパーサ状態の複数の定義を含むと共に、前記パーサ状態の少なくとも１つにおいて実施するための少なくとも１つのプロダクション関数を含むコンパイラ定義ファイルを読み出すステップと、及び
前記コンパイラ定義ファイルを処理して前記第１のコンピュータ言語をコンパイルし、前記第２のコンピュータ言語を生成するステップとからなり、この生成ステップが、
前記第１のコンピュータ言語のステートメントを含むソースファイルを読み出し、前記ソースファイルの前記ステートメントの各々に含まれている各々の記号毎に１つ宛の、複数のトークンを生成するステップと、
前記トークンを構文解析して、前記トークンについて前記複数のパーサ状態の何れを実施すべきかを決定し、前記構文解析によって前記パーサ状態の各々を実施する順序を規定するステップと、
前記複数のパーサ状態の各々の実施時に、実行に備えて識別された前記プロダクション関数を全て実行するステップと、及び

前記パーサ状態をデータ記憶領域へと前記実施順に書き込むステップ
とからなる第１のコンパイルパスを実行するステップと、
前記データ記憶領域から前記パーサ状態を前記実施順に検索するステップと、及び
前記パーサ状態の各々の実施時に、実行に備えて識別された前記プロダクション関数を全て実行し、前記プロダクション関数によって前記第２のコンピュータ言語を生成するステップ
とからなる第２のコンパイルパスを実施するステップとからなり、
前記第１と第２のコンパイルパスの両方を実施するために前記ソースファイルが１回だけ処理される、コンピュータ実施方法。

【０１１６】

２．前記プロダクション関数の各々が、前記第１のコンパイルパス時の実行に備えて識別され、又は前記第２のコンパイルパス時の実行に備えて識別され、或いは前記第１のコンパイルパスと前記第２のコンパイルパスの両方における実行に備えて識別される、上記１のコンピュータ実施方法。

【０１１７】

３．前記第１のコンパイルパス時に実行される前記プロダクション関数の少なくとも１つの実行時にデータをセーブするステップと、前記第２のコンパイルパス時の前記プロダクション関数の実行時に前記セーブされたデータを復元するステップとをさらに含み、前記データのセーブ及び復元が、前記各パス時に前記パーサ状態を実施する前記順序において同じ位置で行われる、上記１のコンピュータ実施方法。

【０１１８】

４．前記データの前記セーブ及び復元が、前記パーサ状態の各々の実施の間に行われる、上記３のコンピュータ実施方法。

【０１１９】

５．前記データの前記セーブ及び復元が、前記コンパイルパスの各々において所定のプロダクション関数が実行される際に行われる、上記３のコンピュータ実施方法。

【０１２０】

６．前記第２のコンパイルパスにおける実行に備えて、前記第１のコンパイルパスにおいて少なくとも１つの関数を識別するステップをさらに含む、上記３のコンピュータ実施方法。

【０１２１】

７．前記第１のコンパイルパス時に前記少なくとも１つの関数に受け渡すべきデータをセーブし、前記第２のコンパイルパスにおいて前記少なくとも１つの関数を実行する際に、前記少なくとも１つの関数に前記データを受け渡すステップをさらに含む、上記６のコンピュータ実施方法。

【０１２２】

８．前記第１のコンパイルパス時に所定のプロダクション関数を実行する場合に、前記第１のコンパイルパスの処理を中断し、前記第２のコンパイルパスを実施して前記データ記憶領域に記憶された前記パーサ状態の全てを前記第２のコンパイルパスで処理し、前記第１のパスの処理を再開するステップをさらに含む、上記１のコンピュータ実施方法。

【０１２３】

９．前記第１のコンパイルパス時に実行される前記プロダクション関数の少なくとも１つの実行時にデータをセーブし、前記第２のコンパイルパス時における前記プロダクション関数の実行時に前記セーブされたデータを復元するステップをさらに含み、前記データが、前記第１のコンパイルパスにおいて前記パーサ状態を実施する前記順序中の第１の位置にセーブされ、前記第２のコンパイルパスにおいて前記パーサ状態を実施する前記順序中の先行位置で復元される、上記１のコンピュータ実施方法。

【０１２４】

１０．第１の所定のプロダクション関数が前記第１のコンパイルパスにおいて実行される際に前記セーブ位置が識別され、第２の所定のプロダクション関数が前記第１のコンパイ

ルパスにおいて実行される際に前記データがセーブされる、上記９のコンピュータ実施方法。

【０１２５】

１１．第１のコンピュータ言語を第２のコンピュータ言語にコンパイルするためのコンピュータ実施方法であって、前記コンパイルが少なくとも２回のコンパイルパスを必要とするものにおいて、前記コンピュータ実施方法が、

前記第１のコンピュータ言語を前記第２のコンピュータ言語にコンパイルするためのパーサ状態の複数の定義を含むコンパイラ定義ファイルを読み出し、前記コンパイラ定義ファイルが前記パーサ状態の少なくとも１つにおいて実施するための少なくとも１つのプロダクション関数を含み、また前記少なくとも１つのプロダクション関数の各々が前記第１のコンパイルパス時の実行に備えて識別され、又は前記第２のコンパイルパス時の実行に備えて識別され、或いは前記第１のコンパイルパスと前記第２のコンパイルパスの両方における実行に備えて識別されるステップと、及び

前記コンパイラ定義ファイルを処理して前記第１のコンピュータ言語をコンパイルし、前記第２のコンピュータ言語を生成するステップとからなり、この生成ステップが、

前記第１のコンピュータ言語のステートメントを含むソースファイルを読み出し、前記ソースファイルの前記ステートメントの各々に含まれている各々の記号毎に１つ宛の、複数のトークンを生成するステップと、

前記トークンを構文解析して、前記トークンについて前記複数のパーサ状態の何れを実施すべきかを決定し、前記構文解析によって前記パーサ状態の各々を実施する順序を規定するステップと、

前記複数のパーサ状態の各々の実施時に前記第１のコンパイルパスにおいて実行に備えて識別された前記プロダクション関数を全て実行し、前記複数のパーサ状態の各々の実施時に前記両方のコンパイルパスにおいて実行に備えて識別された前記プロダクション関数を全て実行するステップと、及び

前記パーサ状態をデータ記憶領域へと前記実施順に書き込むステップ

とからなる前記第１のコンパイルパスを実行するステップと、

前記データ記憶領域から前記パーサ状態を前記実施順に検索するステップと、及び

前記パーサ状態の各々の実施時に前記第２のコンパイルパスにおいて実行に備えて識別された前記プロダクション関数を全て実行し、前記パーサ状態の各々の実施時に前記両方のコンパイルパスにおいて実行に備えて識別された前記プロダクション関数を全て実行し、

前記プロダクション関数によって前記第２のコンピュータ言語を生成するステップ

とからなる第２のコンパイルパスを実施するステップとからなり、

前記第１と第２のコンパイルパスの両方を実施するために前記ソースファイルが１回だけ処理される、コンピュータ実施方法。

【０１２６】

１２．前記第１のコンパイルパス時に実行される前記プロダクション関数の少なくとも１つの実行時にデータをセーブするステップと、前記第２のコンパイルパス時の前記プロダクション関数の実行時に前記セーブされたデータを復元するステップとをさらに含み、前記データのセーブ及び復元が、前記各パス時に前記パーサ状態を実施する前記順序において同じ位置で行われる、上記１１のコンピュータ実施方法。

【０１２７】

１３．前記データの前記セーブ及び復元が、前記パーサ状態の各々の実施の間に行われる、上記１２のコンピュータ実施方法。

【０１２８】

１４．前記データの前記セーブ及び復元が、前記コンパイルパスの各々において所定のプロダクション関数が実行される際に行われる、上記１２のコンピュータ実施方法。

【０１２９】

１５．前記第２のコンパイルパスにおける実行に備えて、前記第１のコンパイルパスにおいて少なくとも１つの関数を識別するステップをさらに含む、上記１１のコンピュータ実

施方法。
【０１３０】
１６．前記第１のコンパイルパス時に前記少なくとも１つの関数に受け渡すべきデータをセーブし、前記第２のコンパイルパスにおいて前記少なくとも１つのプロダクション関数を実行する際に、前記少なくとも１つのプロダクション関数に前記データを受け渡すステップをさらに含む、上記１５のコンピュータ実施方法。
【０１３１】
１７．前記第１のコンパイルパス時に所定のプロダクション関数を実行する場合に、前記第１のコンパイルパスの処理を中断し、前記第２のコンパイルパスを実施して前記データ記憶領域に記憶された前記パーサ状態の全てを前記第２のコンパイルパスで処理し、前記第１のパスの処理を再開するステップをさらに含む、上記１１のコンピュータ実施方法。
【０１３２】
１８．前記第１のコンパイルパス時に実行される前記プロダクション関数の少なくとも１つの実行時にデータをセーブし、前記第２のコンパイルパス時における前記プロダクション関数の実行時に前記セーブされたデータを復元するステップをさらに含み、前記データが、前記第１のコンパイルパスにおいて前記パーサ状態を実施する前記順序中の第１の位置にセーブされ、前記第２のコンパイルパスにおいて前記パーサ状態を実施する前記順序中の先行位置で復元される、上記１１のコンピュータ実施方法。
【０１３３】
１９．第１の所定のプロダクション関数が前記第１のコンパイルパスにおいて実行される際に前記セーブ位置が識別され、第２の所定のプロダクション関数が前記第１のコンパイルパスにおいて実行される際に前記データがセーブされる、上記１８のコンピュータ実施方法。
【０１３４】
２０．第１のコンピュータ言語を第２のコンピュータ言語にコンパイルするためのコンピュータ実施方法であって、前記コンパイルが少なくとも２回のコンパイルパスを必要とするものにおいて、前記コンピュータ実施方法が、
前記第１のコンピュータ言語を前記第２のコンピュータ言語にコンパイルするためのパーサ状態の複数の定義を含むと共に、前記パーサ状態の少なくとも１つにおいて実施するための少なくとも１つのプロダクション関数を含むコンパイラ定義ファイルを読み出すステップと、及び
前記コンパイラ定義ファイルを処理して前記第１のコンピュータ言語をコンパイルし、前記第２のコンピュータ言語を生成するステップとからなり、この生成ステップが、
前記第１のコンピュータ言語のステートメントを含むソースファイルを読み出し、前記ソースファイルの前記ステートメントの各々に含まれている各々の記号毎に１つ宛の、複数のトークンを生成するステップと、
前記トークンを構文解析して、前記トークンについて前記複数のパーサ状態の何れを実施すべきかを決定し、前記構文解析によって前記パーサ状態の各々を実施する順序を規定するステップと、
前記複数のパーサ状態の各々の実施時に、実行に備えて識別された前記プロダクション関数を全て実行するステップと、及び
前記パーサ状態をデータ記憶領域へと前記実施順に書き込むステップ
とからなるスキャンコンパイルパスを実行するステップと、
前記データ記憶領域から前記パーサ状態を前記実施順に検索するステップと、及び
前記パーサ状態の各々の実施時に、実行に備えて識別された前記プロダクション関数を全て実行し、前記プロダクション関数によって前記第２のコンピュータ言語を生成するステップ
とからなる前記スキャンコンパイルパスに後続する少なくとも１つのパスを実施するステップとからなり、
前記スキャンコンパイルパスと前記スキャンコンパイルパスに後続する少なくとも１つの

パスの両方を実施するために前記ソースファイルが１回だけ処理される、コンピュータ実施方法。

【０１３５】

２１．前記プロダクション関数の各々が、前記スキャンコンパイルパス時の実行に備えて識別され、又は前記スキャンコンパイルパスに後続する少なくとも１つのパス時の実行に備えて識別され、或いは前記スキャンコンパイルパスと前記スキャンコンパイルパスに後続する少なくとも１つのパスにおける実行に備えて識別される、上記２０のコンピュータ実施方法。

【０１３６】

２２．前記スキャンコンパイルパス時に実行される前記プロダクション関数の少なくとも１つの実行時にデータをセーブするステップと、前記スキャンコンパイルパスに後続する少なくとも１つのパス時の前記プロダクション関数の実行時に前記セーブされたデータを復元するステップとをさらに含み、前記データのセーブ及び復元が、前記スキャンコンパイルパスに後続する前記パスの各々において前記パーサ状態を実施する前記順序でもって同じ位置で行われる、上記２０のコンピュータ実施方法。

【０１３７】

２３．前記データの前記セーブ及び復元が、前記パーサ状態の各々の実施の間に行われる、上記２２のコンピュータ実施方法。

【０１３８】

２４．前記データの前記セーブ及び復元が、前記コンパイルパスの各々において所定のプロダクション関数が実行される際に行われる、上記２２のコンピュータ実施方法。

【０１３９】

２５．前記スキャンコンパイルパスに後続する前記少なくとも１つのパスにおける実行に備えて、前記スキャンコンパイルパスにおいて少なくとも１つの関数を識別するステップをさらに含む、上記２０のコンピュータ実施方法。

【０１４０】

２６．前記スキャンコンパイルパス時に前記少なくとも１つの関数に受け渡すべきデータをセーブし、前記スキャンコンパイルパスに後続する前記少なくとも１つのパスにおいて前記少なくとも１つの関数を実行する際に、前記少なくとも１つの関数に前記データを受け渡すステップをさらに含む、上記２５のコンピュータ実施方法。

【０１４１】

２７．前記スキャンコンパイルパス時に所定のプロダクション関数を実行する場合に、前記スキャンコンパイルパスの処理を中断し、前記スキャンコンパイルパスに後続する前記少なくとも１つのパスを実施して前記データ記憶領域に記憶された前記パーサ状態の全てを処理し、前記スキャンコンパイルパスの処理を再開するステップをさらに含む、上記２０のコンピュータ実施方法。

【０１４２】

２８．前記スキャンコンパイルパス時に実行される前記プロダクション関数の少なくとも１つの実行時にデータをセーブし、前記スキャンコンパイルパスに後続する前記少なくとも１つのパス時における前記プロダクション関数の実行時に前記セーブされたデータを復元するステップをさらに含み、前記データが、前記スキャンコンパイルパスにおいて前記パーサ状態を実施する前記順序中の第１の位置にセーブされ、前記スキャンコンパイルパスに後続する前記少なくとも１つのパスにおいて前記パーサ状態を実施する前記順序中の先行位置で復元される、上記２０のコンピュータ実施方法。

【０１４３】

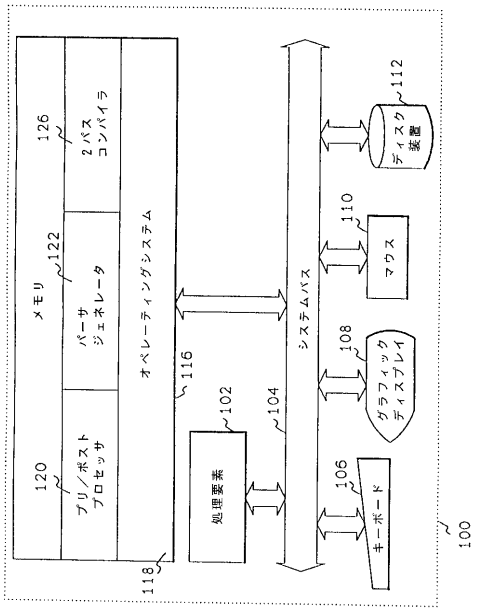
２９．第１の所定のプロダクション関数が前記スキャンコンパイルパスにおいて実行される際に前記セーブ位置が識別され、第２の所定のプロダクション関数が前記スキャンコンパイルパスにおいて実行される際に前記データがセーブされる、上記２８のコンピュータ実施方法。

【０１４４】

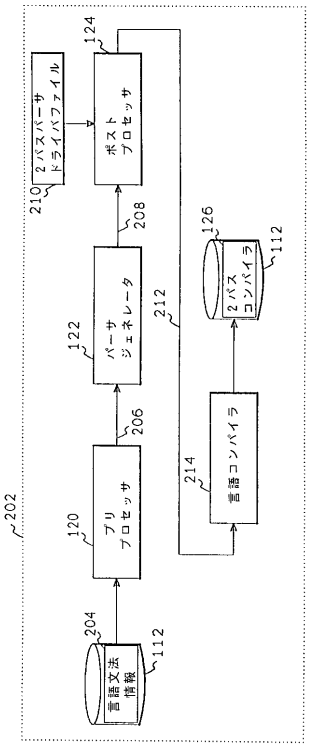
【発明の効果】
本発明は上述のように構成したので、ソース入力を１回しか処理しない高効率の２パス言語コンパイラを提供することができる。
【図面の簡単な説明】
【図１】本発明を組み込んだコンピュータシステムを示すブロック図である。
【図２】本発明の２パスコンパイラを作成するプロセスを示すブロック図である。
【図３】２パスコンパイラの操作を示すブロック図である。
【図４】図２のプリプロセッサブロックおよびポストプロセッサブロックを示すブロック図である。
【図５】図４のプリプロセッサパーサを示すフローチャートである。
【図６】図４のポストプロセッサパーサを示すフローチャートである(1/2)。
【図７】図４のポストプロセッサパーサに関するフローチャートである(2/2)。
【図８】図５から呼び出される関数yyparseを示すフローチャートである(1/2)。
【図９】図５から呼び出される関数yyparseを示すフローチャートである(2/2)。
【図１０】図４のプリ／ポストプロセッサスキャナを示すフローチャートである(1/3)。
【図１１】図４のプリ／ポストプロセッサスキャナを示すフローチャートである(2/3)。
【図１２】図４のプリ／ポストプロセッサスキャナを示すフローチャートである(3/3)。
【図１３】図２の２パスコンパイラ126の動作を示すブロック図である。
【図１４】図２の２パスコンパイラ126のためのパーサを示すフローチャートである(1/3)。
【図１５】図２の２パスコンパイラ126のためのパーサを示すフローチャートである(2/3)。
【図１６】図２の２パスコンパイラ126のためのパーサを示すフローチャートである(3/3)。
【図１７】２パスコンパイラで用いるための利用可能な関数twopass_saveを示すフローチャートである。
【図１８】２パスコンパイラで用いるための利用可能な関数twopass_restoreを示すフローチャートである。
【図１９】２パスコンパイラで用いるための利用可能な関数twopass_callfuncを示すフローチャートである。
【図２０】図１９から呼び出される関数yy_syncvarを示すフローチャートである。
【図２１】図１４および図１６から呼び出される書込モード切換関数を示すフローチャートである。
【図２２】図１５から呼び出される読出モード切換関数を示すフローチャートである。
【図２３】図１４から呼び出される状態書込関数を示すフローチャートである。
【図２４】図２３から呼び出される関数yywriteを示すフローチャートである。
【図２５】図１５および図１６から呼び出される関数yyreadを示すフローチャートである。
【図２６】２パスコンパイラで利用可能な関数twopass_save_locを示すフローチャートである。
【図２７】２パスコンパイラで利用可能な関数twopass_stuff_locを示すフローチャートである。
【図２８】２パスコンパイラで利用可能な関数twopass_dopass2を示すフローチャートである。
【符号の説明】
126　２パスコンパイラ
302　ソースコードファイル
304　字句スキャナ
306　第１パスパーサ
308　第２パスパーサ

310　パーサ動作関数
312　中間ファイル

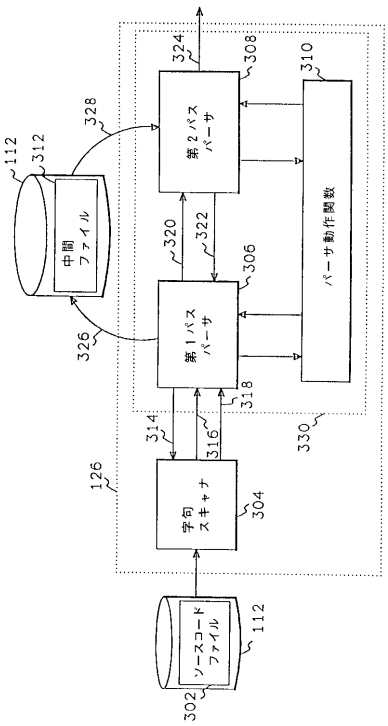
【図１】



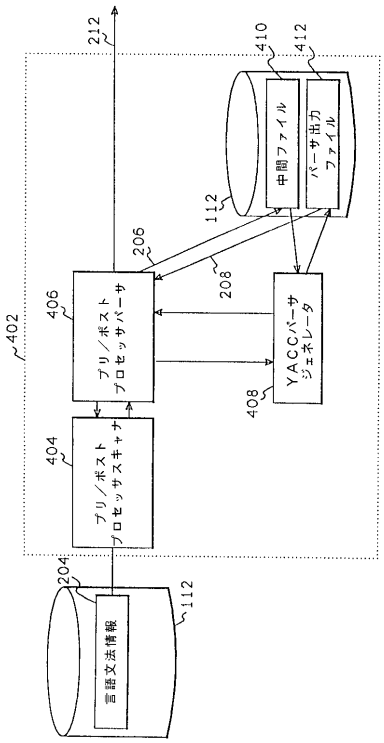

【図２】



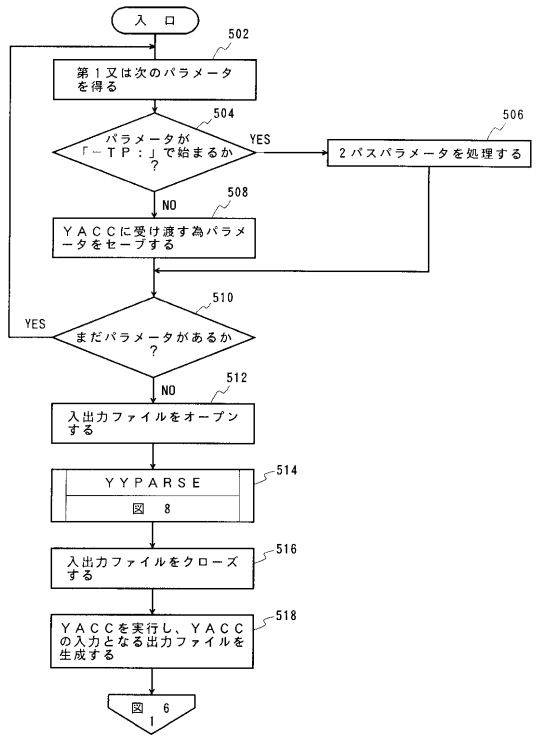
【図３】


第２パス パーサ
第１パス パーサ
字句 スキャナ
パーサ動作関数
中間 ファイル
ソースコード ファイル
112
126
302
304
306
308
310
312
314
316
318
320
322
324
326
328
330

【図４】


プリ／ポスト プロセッサパーサ
プリ／ポスト プロセッサスキャナ
YACCパーサ ジェネレータ
中間ファイル
パーサ出力 ファイル
言語文法情報
112
204
205
206
208
212
402
404
406
408
410
412

【図５】


入 口
第１又は次のパラメータを得る
パラメータが「－ＴＰ：」で始まるか？
YES
NO
２パスパラメータを処理する
ＹＡＣＣに受け渡す為パラメータをセーブする
まだパラメータがあるか？
入出力ファイルをオープンする
ＹＹＰＡＲＳＥ
図 8
入出力ファイルをクローズする
ＹＡＣＣを実行し、ＹＡＣＣの入力となる出力ファイルを生成する
図 6
1
502
504
506
508
510
512
514
516
518

【図６】

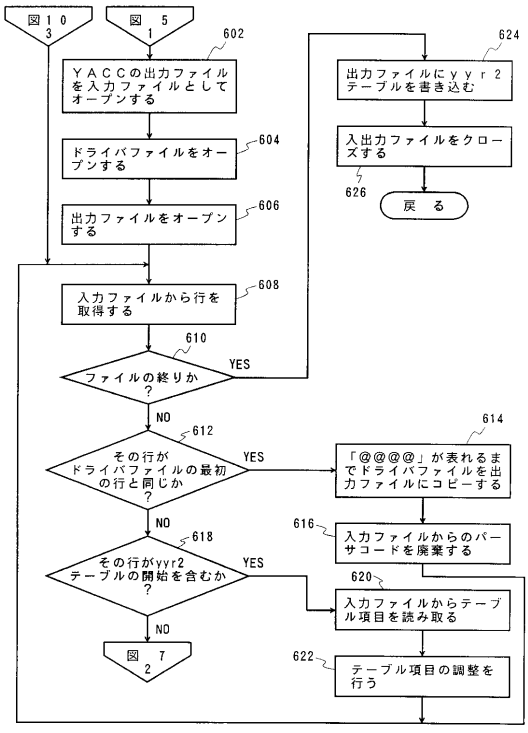


図１０ 3
図 5 1
ＹＡＣＣの出力ファイルを入力ファイルとしてオープンする
ドライバファイルをオープンする
出力ファイルをオープンする
入力ファイルから行を取得する
ファイルの終りか？
YES
NO
その行がドライバファイルの最初の行と同じか？
「@@@@」が表れるまでドライバファイルを出力ファイルにコピーする
入力ファイルからのパーサコードを廃棄する
その行がyyr2テーブルの開始を含むか？
入力ファイルからテーブル項目を読み取る
テーブル項目の調整を行う
図 7 2
出力ファイルにｙｙｒ２テーブルを書き込む
入出力ファイルをクローズする
戻 る
602
604
606
608
610
612
614
616
618
620
622
624
626

【図７】

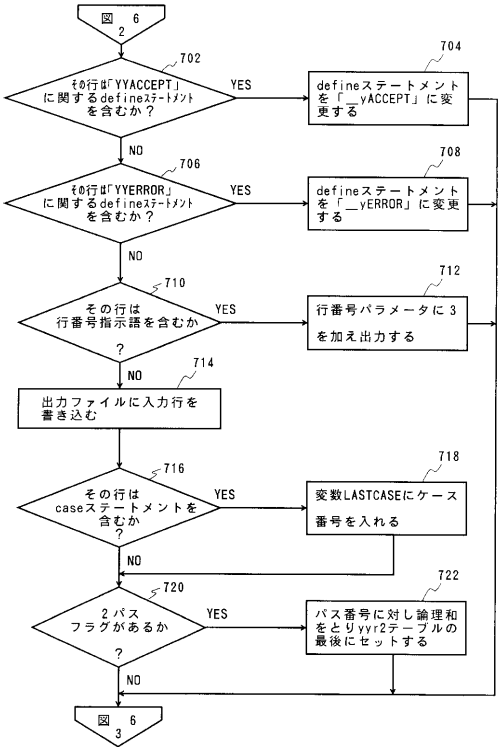


図 6
2
その行は「YYACCEPT」に関するdefineステートメントを含むか？
YES
defineステートメントを「_yACCEPT」に変更する
NO
その行は「YYERROR」に関するdefineステートメントを含むか？
YES
defineステートメントを「_yERROR」に変更する
NO
その行は行番号指示語を含むか？
YES
行番号パラメータに3を加え出力する
NO
出力ファイルに入力行を書き込む
その行はcaseステートメントを含むか？
YES
変数LASTCASEにケース番号を入れる
NO
2パスフラグがあるか？
YES
パス番号に対し論理和をとりyyr2テーブルの最後にセットする
NO
図 6
3

【図８】

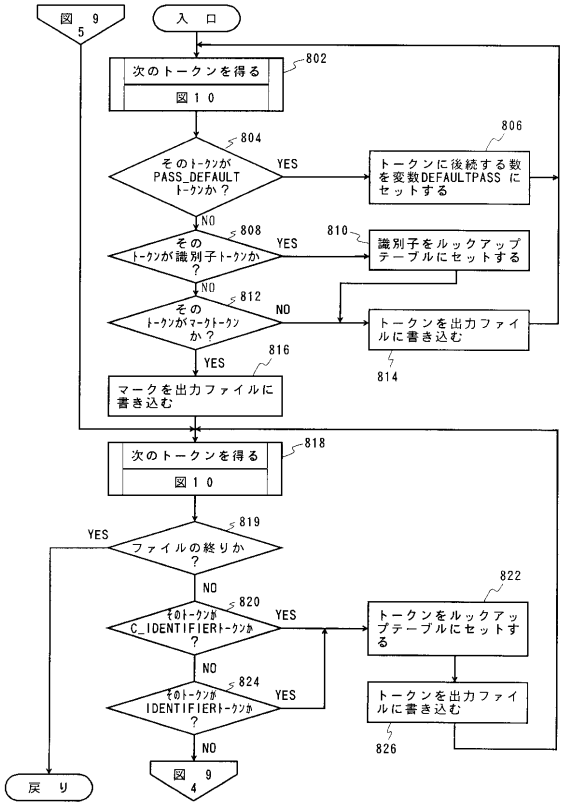


図 9
5
入 口
次のトークンを得る
図１０
そのトークンがPASS_DEFAULTトークンか？
YES
トークンに後続する数を変数DEFAULTPASSにセットする
NO
そのトークンが識別子トークンか？
YES
識別子をルックアップテーブルにセットする
NO
そのトークンがマークトークンか？
NO
トークンを出力ファイルに書き込む
YES
マークを出力ファイルに書き込む
次のトークンを得る
図１０
ファイルの終りか？
YES
NO
そのトークンがC_IDENTIFIERトークンか？
YES
トークンをルックアップテーブルにセットする
NO
そのトークンがIDENTIFIERトークンか？
YES
トークンを出力ファイルに書き込む
NO
図 9
4
戻 り

【図９】

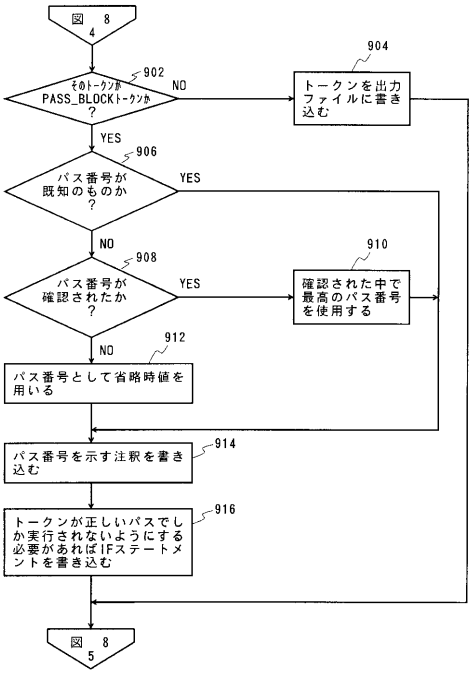
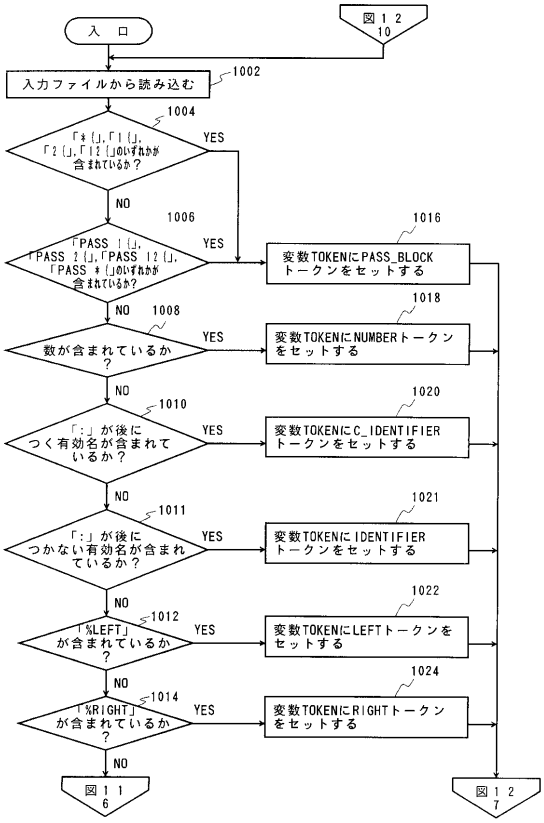
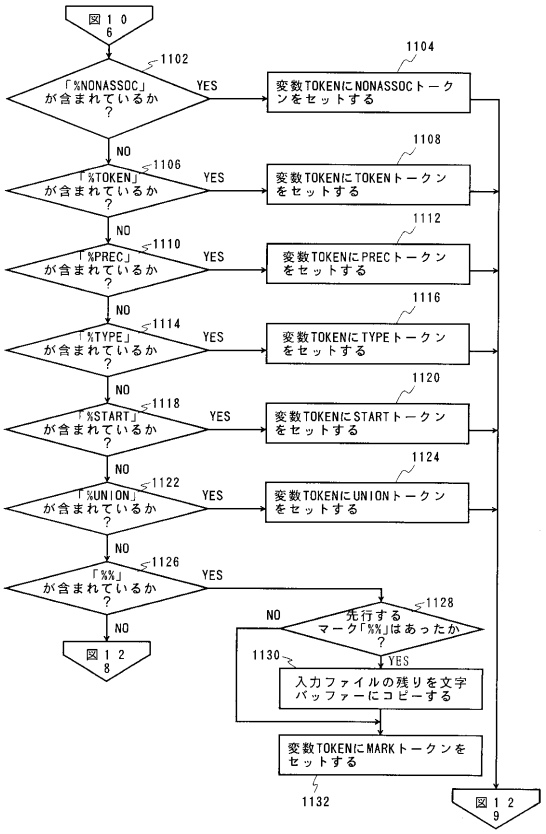


図 8
4
そのトークンがPASS_BLOCKトークンか？
NO
トークンを出力ファイルに書き込む
YES
パス番号が既知のものか？
YES
NO
パス番号が確認されたか？
YES
確認された中で最高のパス番号を使用する
NO
パス番号として省略時値を用いる
パス番号を示す注釈を書き込む
トークンが正しいパスでしか実行されないようにする必要があればIFステートメントを書き込む
図 8
5

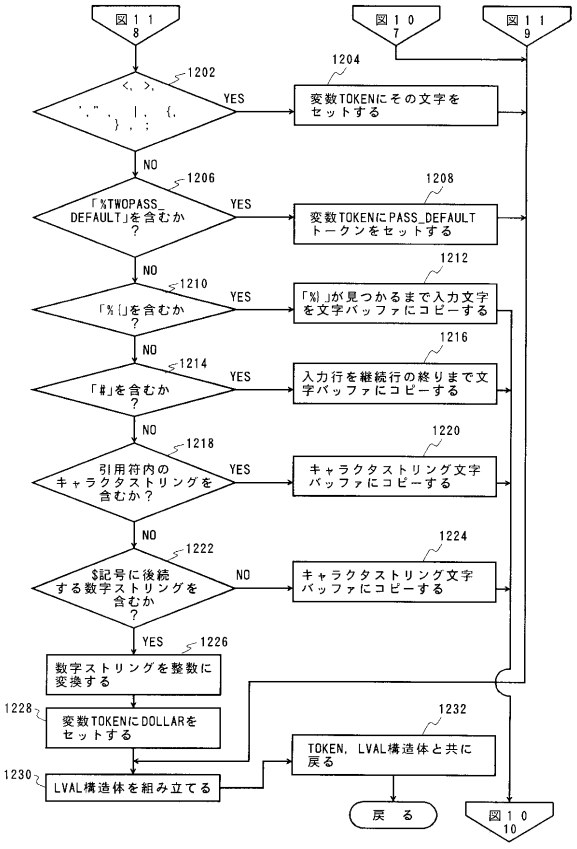
【図１０】


入 口
図１２
10
入力ファイルから読み込む
「¥{」,「!{」,「?{」,「!?{」のいずれかが含まれているか？
YES
NO
「PASS !{」,「PASS !?{」,「PASS ¥{」のいずれかが含まれているか?
YES
変数TOKENにPASS_BLOCKトークンをセットする
NO
数が含まれているか？
YES
変数TOKENにNUMBERトークンをセットする
NO
「:」が後につく有効名が含まれているか？
YES
変数TOKENにC_IDENTIFIERトークンをセットする
NO
「:」が後につかない有効名が含まれているか？
YES
変数TOKENにIDENTIFIERトークンをセットする
NO
「%LEFT」が含まれているか？
YES
変数TOKENにLEFTトークンをセットする
NO
「%RIGHT」が含まれているか？
YES
変数TOKENにRIGHTトークンをセットする
NO
図１１
6
図１２
7

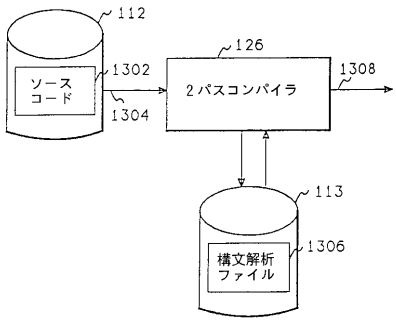
【図１１】



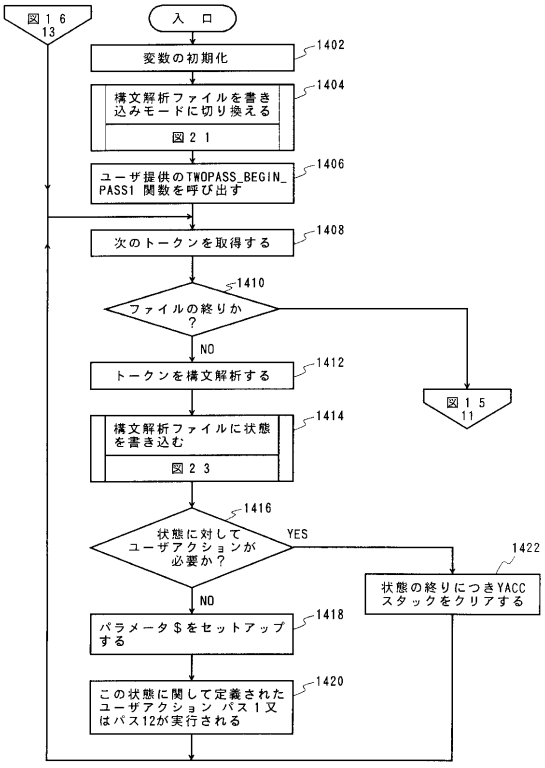

【図１２】



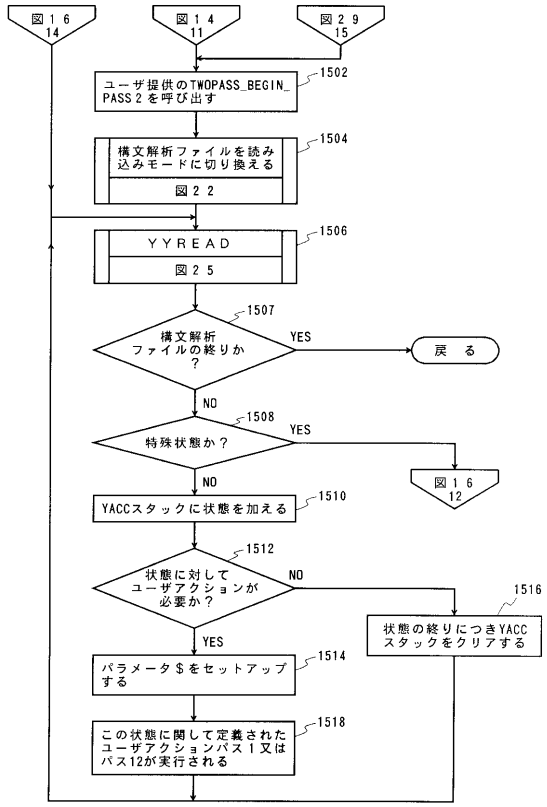

【図１３】



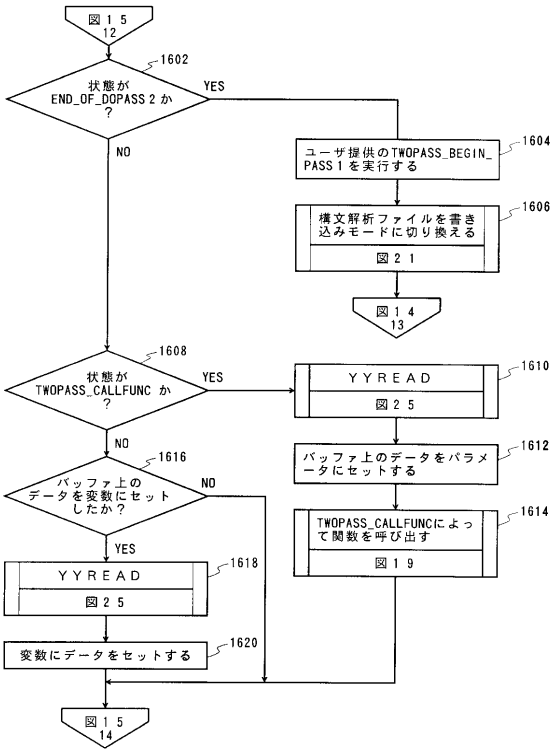

【図１４】



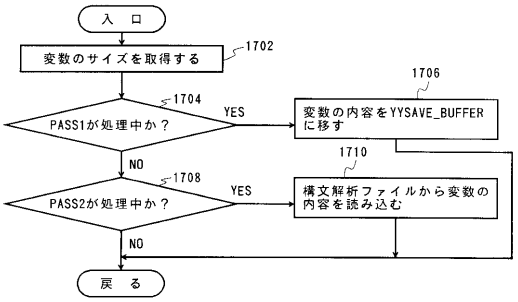
【図１５】



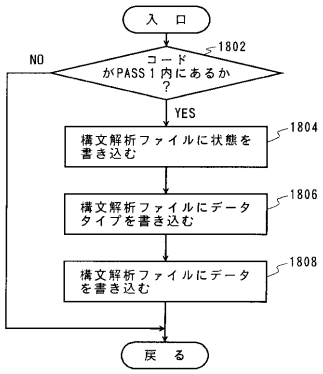

【図１６】



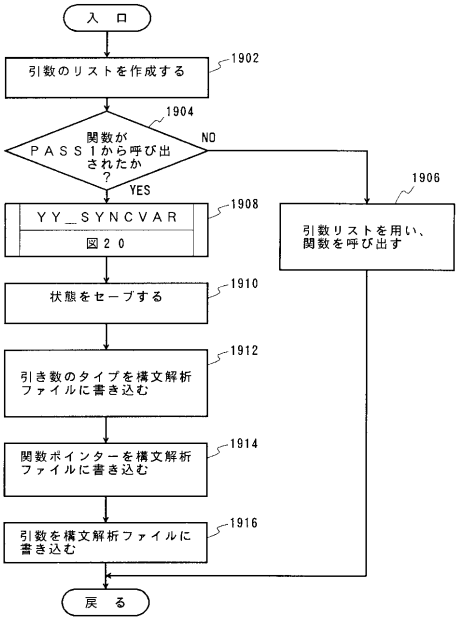

【図１７】



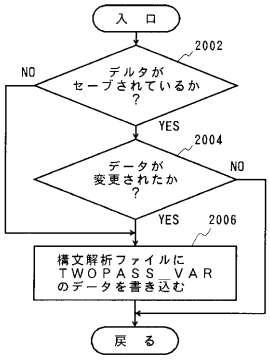

【図１８】



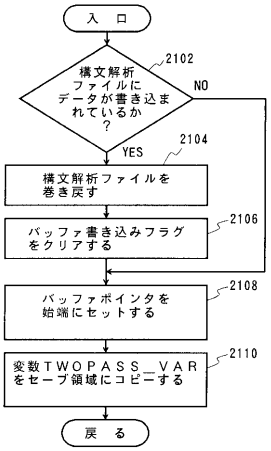

【図１９】



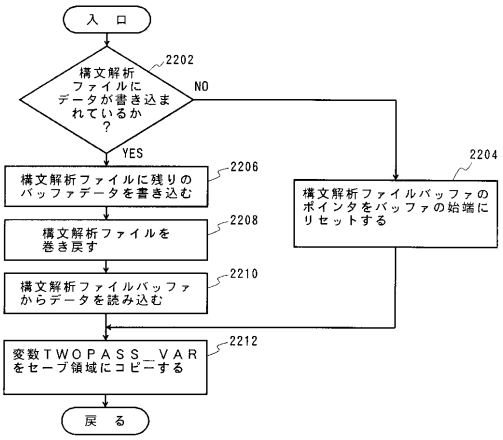
【図２０】



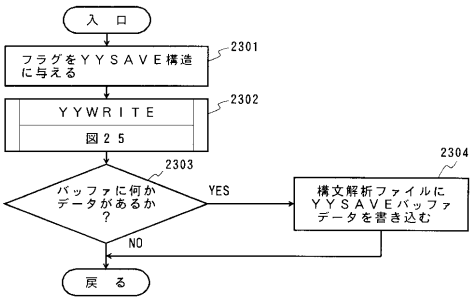

【図２１】



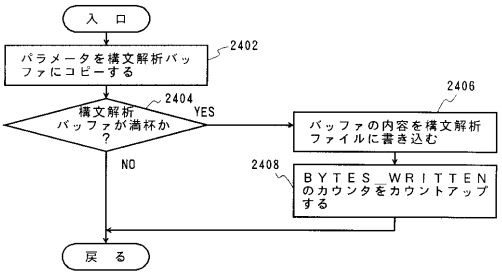

【図２２】



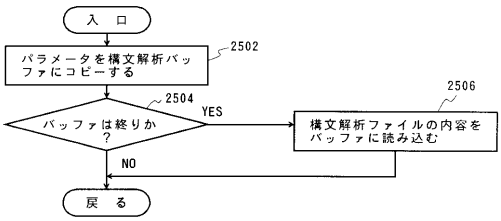

【図２３】



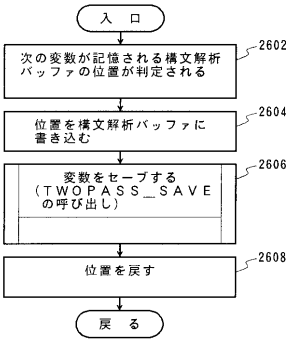

【図２４】



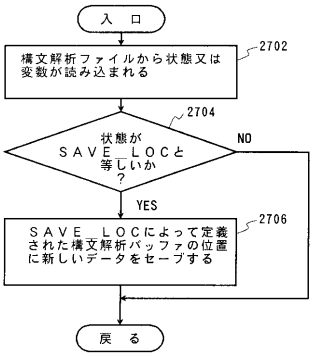

【図２５】



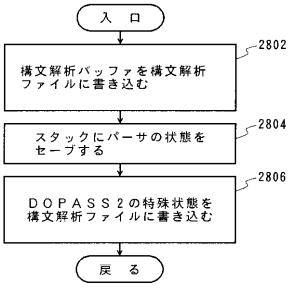

【図２６】




【図２７】



【図２８】


入　口
構文解析バッファを構文解析ファイルに書き込む
2802
スタックにパーサの状態をセーブする
2804
ＤＯＰＡＳＳ２の特殊状態を構文解析ファイルに書き込む
2806
戻　る

フロントページの続き

(72)発明者 グレゴリー・エス・リンドホルスト
アメリカ合衆国カリフォルニア州９８０７２ウッディンヴィレ，ノースイースト・ワンハンドレッドナインティセヴンス・プレイス・１３０３３

審査官 久保 光宏

(56)参考文献 特開平５－１４３３５７（ＪＰ，Ａ）
吉田正和，「ソフトウェア・ツールの内側 １．コンパイラのしくみ」，インターフェース，日本，ＣＱ出版株式会社，１９９１年 ８月 １日，Vol.17，No.8，pp.124-142，ISSN:0387-9569
Mossenbock，H.，"A Convenient Way to Incorporate Semantic Actions in Two-pass Compiling Schemes"，SOFTWARE-PRACTICE AND EXPERIENCE，１９８８年 ７月，Vol.18，No.7，pp.691-700，JST資料番号:E0674A，ISSN:0038-0644
Levine，J.R.（岩谷宏・訳），「Ｃプログラミングの中でのlexとyaccの使い方」，C MAGAZINE，日本，ソフトバンク株式会社，１９９３年 ４月 １日，Vol.5，No.4，pp.27-29
「日立エンジニアリングワークステーション 2050G HI-UX/GE HI-UXプログラマーズガイド 205G-3-603」，日本，株式会社日立製作所，１９９２年 ２月，初版，pp.159-232

(58)調査した分野(Int.Cl.[7]，ＤＢ名)
G06F9/45，
G06F9/06，
JSTファイル(JOIS)，
CSDB(日本国特許庁)，
INSPEC(DIALOG)，
WPI/L(DIALOG)