P1　コンパイラの作成

1　目的

簡単なプログラミング言語のコンパイラを作成することにより，コンパイラの構造，プログラミング言語，および計算機アーキテクチャについて理解を深めると共に，ある程度大きなプログラム作成を通じてソフトウェアシステム構築技術を習得する。

2　コンパイラの基礎

C言語やPascalなどの高級言語は，人間に分かりやすい書き方でプログラムを書ける計算機言語である。しかし，そのままでは計算機が実行できる機械語ではないため，機械語に変換する操作が必要である。この変換を行うプログラムをコンパイラという。コンパイラに入力する高級言語で書かれたプログラムを原始プログラム，コンパイラが出力する機械語あるいはそれに近い低レベルの言語で書かれたプログラムを目的プログラム，目的プログラムを最終的に実行させる計算機を目的計算機と呼ぶ。一般に計算機にはアセンブラが存在するので，コンパイラの目的プログラム言語をアセンブラ言語とし，アセンブラが実際の機械語に変換することが多い。本実験で作成するコンパイラも，目的プログラム言語をアセンブラ言語とし，アセンブラ言語のプログラムから機械語プログラムへの変換は，こちらで用意するアセンブラで行う。

コンパイラはいくつかのフェーズからなる。フェーズとは処理の論理的な一つのまとまりであり，原始プログラムからフェーズ毎に徐々に目的プログラムへ近づいていくと考えることができる。単純なコンパイラの典型的なフェーズの構成を図1に示す。四角の中が各フェーズである。以下，簡単に各フェーズの説明を行う。

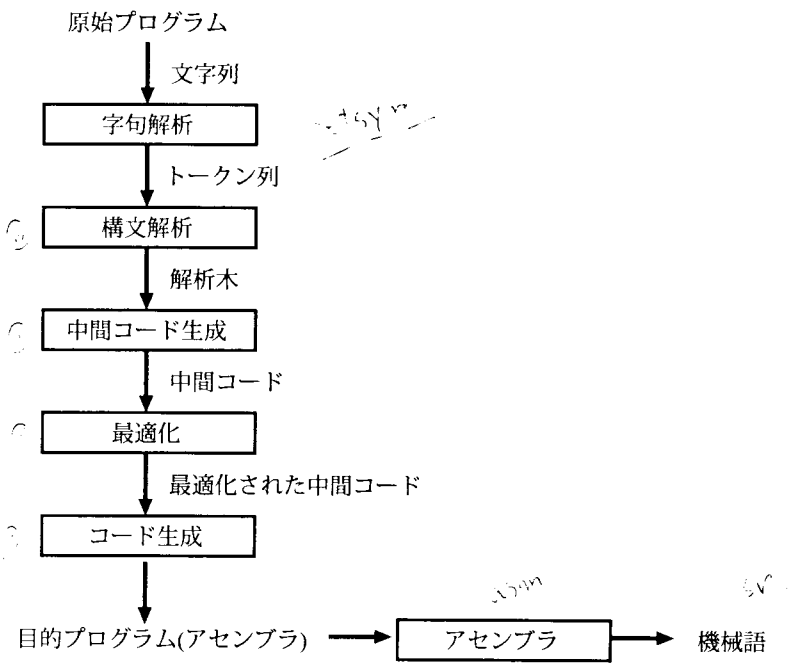


図 1: コンパイラの構成

2.1 字句解析

原始プログラムは人間が見ると，いくつかの意味のある最小の単位 (文字がいくつか集まったもの) から構成されていることが自然と分かる。例えば，図 2 のような C 言語のプログラムを考える。文字がいくつか集まった最小の単位は，「main」，「(」，「)」，「{」，「int」，「i」，「,」，「;」...などである。人間であれば容易にこの最小単位を認識できるが，計算機が読み込む原始プログラムは計算機から見ると単なる文字を表す数字の列であり，工夫をしなければ自然とこの最小単位に分解してくれる訳ではない。この最小単位に分ける処理を字句解析と呼び，主に次のような 2 つの処理を行う。

- 最小単位の区切りを解析する。

  例えば，「sum3 = 0;」という文字列に関して，「sum3」と「=」と「0」と「;」で区切れるということを解析する。

- 最小単位が表している意味を解析する。

  例えば，「sum3 = 0;」という文字列に関して，「sum3」は変数名や関数名を表す文字列，「=」は代入記号，「0」は数字，「;」は 1 つの文の終わりを表す記号であることを解析する。変数名についている数字の「3」は，あくまでも変数名などの名前の一部であり，「0」のような数字とは異なることを解析する。

本実験ではこちらで用意した字句解析プログラムを使用する。この字句解析プログラムは原始プログラムを読み込んで，自動的に意味のある最小単位を順番に切り出し，その最小単位の情報を持つトークンと呼ばれるデータ構造の列に変換する (実際には一回呼び出すごとに 1 つの最小単位を切り出し，1 つのトークンを返す)。この字句解析プログラムの詳細は 付録 C に付ける。

2.2 構文解析・中間コード生成

上記の字句解析プログラムで最小単位に分解された原始プログラムは，構文解析によって意味のある単位に階層的にまとめられる。例えば，図 2 の例のプログラムで構文解析された大まかな結果を見てみると，図 3 のように，全体が main と呼ばれる関数であり，その中で変数の宣言部，代入文，for 文などによって階層的に構成されていることが分かる。for 文は，さらに代入文，条件文などから構成されている。さらに，一つの代入文は図 4 のように階層的に構成されている。

このように，一つの意味のある大きな単位に最小単位から階層的にまとめるフェーズを，構文解析と呼ぶ。構文解析の結果から単にプログラムの構造が分かるだけでなく，まとめた単位がその単位を含むより大きな単位の中でどのような意味をもっているのかについても，知ることができる。例えば図 4 の例では，全体が「代入文」であることから，その一部である「数式」の部分の計算結果を「変数」の中に代入する目的プログラムを出力するように，コンパイラを作ることができる。さらに，「数式」と解析された部分に関しては，「sum3」という「変数」の値と「i」という「変数」の値を加算するような，目的プログラムを出力すればよい。このように，字句解析で得られた最小単位の役割が構文解析によって解析され，その役割に応じたコンパイラの内部ルー

```
main() {
    int i, sum3;
    sum3 = 0;
    for (i = 0; i < 10; i++) {
        sum3 = sum3 + i;
    }
}
```

図 2: C 言語のプログラム

チンによって，目的プログラム，あるいは中間コードが生成される。中間コードとは，目的プログラムに変換する前に最適化などを簡単に行うために，構文解析の結果などの情報を保持している中間的なコードである。最適化などを行わない場合は，構文解析フェーズから直接，目的プログラムを生成することもある。本実験では，数式に関しては中間コードを用いるが，その他では構文解析フェーズから直接目的プログラムを生成する。

2.3 最適化・コード生成

中間コードから直接コード生成を行い，目的プログラムを生成してもかまわないが，構文解析・中間コード生成フェーズからの出力は効率的でない場合が多い。このため，目的プログラムをより速く実行できるように，必要でない部分を削ったり，実行順序の入れ替えなどを行う。これが最適化のフェーズである。本実験では特にこの部分を作成することを義務とはしないが，興味がある学生は作成した方がよい。

最適化が終わった中間コードを目的プログラムに変換するフェーズを，コード生成と呼ぶ。これは，機械的に目的プログラムに置き換えることによってなされる。構文解析の説明で述べたように，本実験では数式以外の部分は構文解析と同時に目的プログラムを生成するので，コード生成のフェーズは数式の部分のコンパイルを行う場合のみである。また，最適化も行わないのであれば，数式の部分も中間コードを生成せず，直接，目的プログラムを生成してもよい。

3 各作成実験の解説

3.1 作成実験の概要

3.1.1 作成するコンパイラの概要

4段階の作成実験を通して，最終的に次のようなコンパイラを生成する。

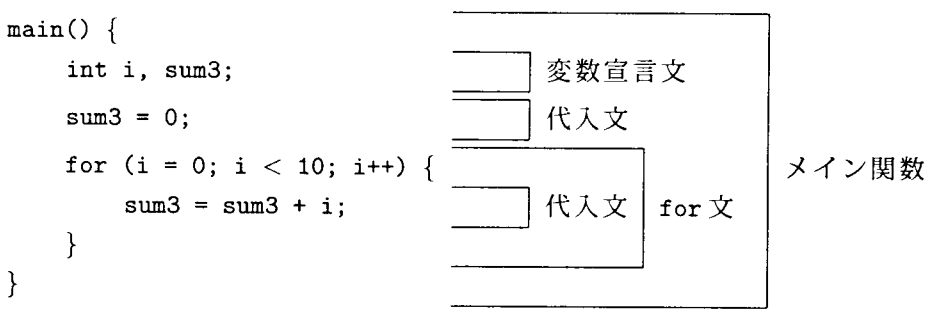


図 3: C 言語のプログラムの構造

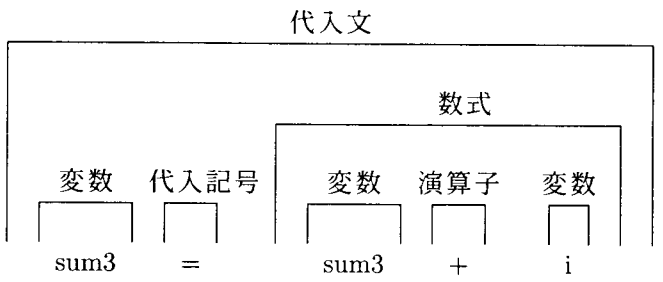


図 4: 代入文の構造

原始プログラムの言語 (構文・意味) 原始プログラムの言語に関しては，作成実験の段階によって，徐々により複雑な入力言語を処理できるように，拡張していく方法を取る。このため，各作成実験に応じた原始プログラムの構文，意味が与えられる。おおよそ，PL/0と呼ばれるPascalに似た言語のサブセット $+\alpha$ である。最終的には次のような言語を受理するコンパイラとなる。

- PASCALのサブセット (PL/0)$+\alpha$
- データ型は整数型のみ
- 再帰呼び出し可能とする
- 手続き定義のネストはない
- 制御文は `if` 文，`while` 文，複合文のみ
- 手続きの引数は，値渡し

目的計算機の仕様の概略 目的計算機のアーキテクチャ，および目的プログラムの言語は全ての段階の作成実験で共通であるが，使用する目的プログラムの命令は徐々に多くなっていく。詳しくは 付録A，付録Bにそれぞれ解説している。概略は次の通りである。

- レジスタマシン

汎用レジスタ	4個
ベースレジスタ	1個
スタックポインタ	1個
基本命令	15種

コンパイラ記述言語 コンパイラ記述言語とは，コンパイラを作るために使用する言語のことである。本実験では原則的にC言語を考えており，C言語で作成するためにいくつかのサンプルプログラムなどを用意してあるが，他の言語，例えばLISPなどの言語でも作成してもよい。しかしその場合，サンプルプログラムなどのサポートは完全に行われないので，自分の責任ですべてを作らなければならない。

3.1.2 実験スケジュール

作成実験1：「簡単なコンパイラの作成と実験ツール」(第1週) 非常に簡単な言語のコンパイラを作成し，実験のためのツールに慣れ，コンパイラの概要，ターゲットマシンと命令セット，字句解析，コード生成についての大まかな知識を修得する。

作成実験2：「変数の扱いと `if` 文，`while` 文」(第2週 – 第3週) 変数，`if` 文，`while` 文を含む言語のコンパイラへと，作成実験1で作成したコンパイラを拡張する。

作成実験3：「数式の翻訳と構文解析」(第4週 – 第5週) 一般的な数式を含む言語のコンパイラへの拡張を行う。

作成実験4：「実行モデルと手続き」(第6週 – 第8週) 手続き定義と手続き呼び出しができる言語のコンパイラへの拡張を行う。

3.1.3 実験グループ

コンパイラは1人で1つ作成する。グループでの開発は認めない。

例 1:

```
program example1;
526 * 321.
```

例 2:

```
program example2;
begin
    100 + 21;
    30 - 643;
    10 * 32
end.
```

例 3:

```
program example3;
begin
    begin
        200 + 10;
        30 - 8
    end;
    13 * 46
end.
```

図 5: 作成実験 1 の原始プログラム例

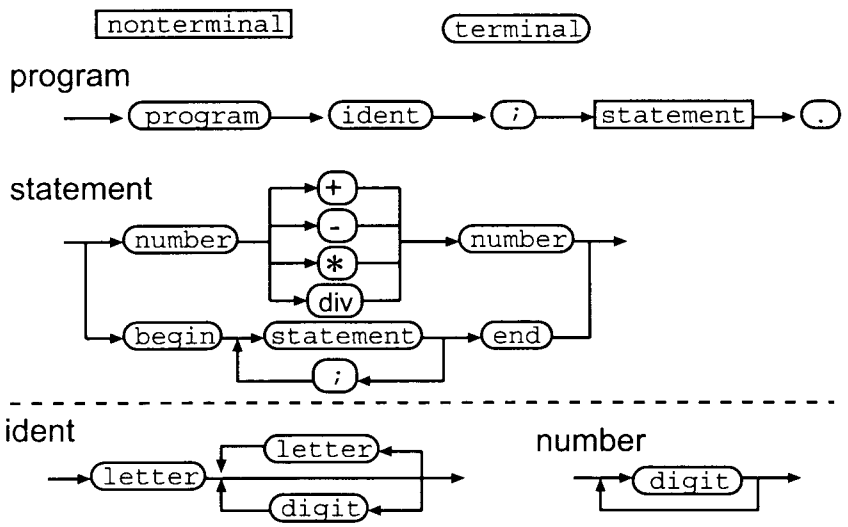


図 6: 作成実験 1 の構文図

3.2 作成実験 1 「簡単なコンパイラの作成と実験ツール」

この段階では非常に簡単な構文を持つ言語のコンパイラを作成し，コンパイラの概要，実験ツールの使用法の修得をめざす。おおよそ，1 回の説明と 2 回の演習で完了する予定である。

3.2.1 原始プログラムの構文と意味論

作成実験 1 で作成するコンパイラの原始プログラムの例を図 5，構文図を図 6 に示す。構文図は，コンパイラが目的プログラムに変換できる原始プログラムの「形」を正確に規定する。日本語で言えば日本語文法にあたるが，計算機の言語は自然言語のような曖昧さは許さない。構文図で規定された形以外のプログラムは，文法誤りとしてリジェクトされる。また原始プログラムの規定としてもう一つ重要なものに，そのプログラムが意味することを規定する「意味論」がある。例えば，「`r = 3 * x`」という式があった場合，C 言語のプログラマならば，「`3`」と「`x`」という変数の値をかけたものを「`r`」という変数に格納すると考えるが，計算機言語を全く知らない人は「`*`」の記号を乗算だとは思わないかもしれない。

『「`*`」は乗算の記号である』という規定が意味論である。しかし，意味論を正確に記述することは困難であるので，本実験では本指導書の中で注意するところだけ日本語で与える。特に，規定されていない部分は常識的に解釈して欲しい。すなわち，「`*`」はかけ算の演算子であり，わり算

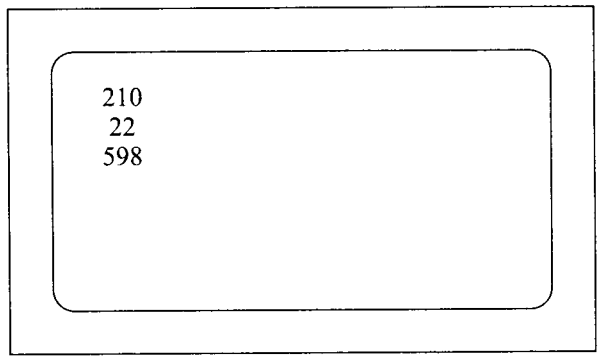


図 7: プログラムの出力例

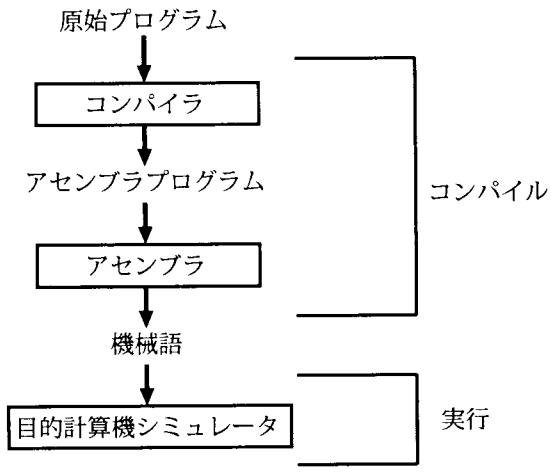


図 8: コンパイルと実行の手順

の演算子ではない。

図 6 の構文図の意味論は次の日本語文で規定される。特に規定されていない部分は常識的に解釈すること (この場合は，特に何もしないということになる)。

図 6 の意味論：「数式 (expression) を計算し，数式が現れる順番にその結果をディスプレイに表示する。表示する結果は各数式の結果ごとに改行を行い，1 行に 1 つの結果を表示する。」

すなわち，図 6 の例 3 のプログラムの実行結果は図 7 のようになる。

3.2.2 実験ツール

2 つの実験ツールをこのプロジェクト実験を通して使用する。機械語シミュレータとアセンブラである。機械語シミュレータは，本実験で仮定している計算機の機械語を，実験で用いるコンピュータ上でシミュレーションするために使用する。実験で用いる実際の計算機の機械語は非常に複雑であり，それを学習するだけで時間がかかるため，このような方法を取る。アセンブラは，本実験で作成するコンパイラが出力する目的プログラム (アセンブラ言語で書かれている) をアセンブルして，上記の機械語シミュレータで実行できる機械語に変換するために使用する。実際に諸君が作成するコンパイラを用いて，原始プログラムを実行するまでの手順を図 8 に示す。アセンブラと機械語シミュレータの詳しい使い方は，付録 A，付録 B にそれぞれ解説する。

3.2.3 実際の作成

付録Eに，statementを除いた部分の構文解析だけを行うC言語のプログラムを示す。このプログラムでは字句解析に，付録Cに詳しく説明するルーチンを用いている。字句解析ルーチンから得られた1つのトークンを，C言語のif文で判断することにより，図6の構文図をたどっていくように作られている。構文図で可能性がないトークンが来たら，エラーを出力するようになっている。構文図の独立した各部分をC言語の関数で記述することによって，「begin...end」のネストのような再帰的な構文も処理できる。付録Eを参考にstatementを解析するルーチンを追加すれば，構文図をたどり，構文解析するルーチンができあがる。さらに，構文の意味に従い，アセンブラプログラムを出力するように改良すれば，簡単なコンパイラができる。

作成実験1の意味論では，「数式」すなわちexpressionにだけ意味があるから，expressionを構文解析すると同時に，「計算を行い，結果をディスプレイに表示するアセンブラプログラム」を出力すればよい。例えば，図9のような変換がなされればよい。ここで，出力のアセンブラプログラムの各部分は図9に示したように，原始プログラムの数式に対応している。原始プログラムに数式が複数出てくる場合は，数式を解析するコンパイラのルーチンを複数回通る。したがって，数式を解析するルーチンは，一つの数式のアセンブラプログラムを正確に出力するように作成すればよい。

数式を解析するルーチンは，構文図に従えば，数字(number)を見て，次に演算子を見，最後にまた数字を見るはずである。アセンブラプログラムの方では，まず1つ目の数字をあるレジスタにロード(loadi)し，次にそのレジスタと最後の数字を演算子が意味する演算で処理(addiなど)すればよい。このように2つの命令で構成されるが，2つ目の命令は演算子と最後の数字を見た後でしか出力できない。このため，演算子を覚えておいて，最後の数字を見た時点で2つ目の演算を行う命令を出力する。最後に，原始プログラムの意味に従って，計算結果が格納されているレジスタの内容をwrited命令で出力し，loadi, writec命令で改行するようなアセンブラプログラムを出力すればよい。

3.3 作成実験2「変数の扱いとif文，while文」

3.3.1 原始プログラムの構文と意味論

作成実験2では，変数，if文，while文，write文を扱えるように，作成実験1のコンパイラを拡張する。作成実験2で作成するコンパイラの原始プログラムの例を図10に，構文図を図11に示す。ただし，図11の点線で囲まれている部分は関係ない(作成実験4で実現する)。write文の導入に伴い，数式の意味としてディスプレイに表示することは省かれ，数式はその値を計算すると

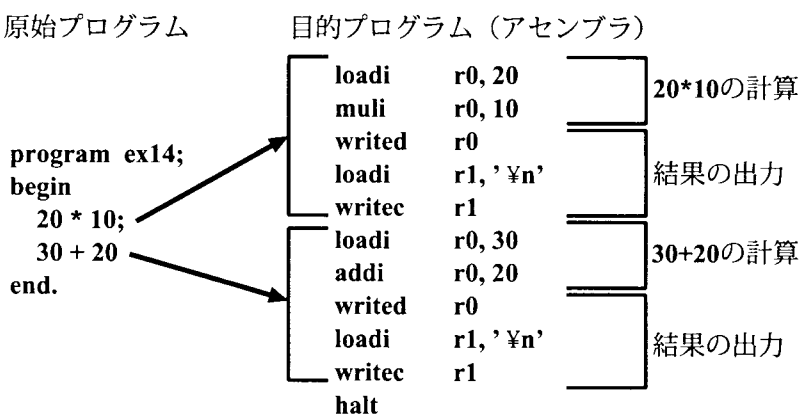


図9: 作成実験1のコンパイル例

例 1:

```
program ex21;
var x,y;
begin
    x := 10;
    y := 2 * x;
    write x, y
end.
```

例 2:

```
program ex22;
var x, y, z;
begin
    x := 100;
    y := 200;
    if y - x >= 50 then z := 2 * x
                   else z := 3 * x;
    write x, y, z
end.
```

例 3:

```
program ex23;
var i, sum;
begin
    i := 0;
    sum := 0;
    while i <= 10 do
        begin
            sum := sum + i;
            i := i + 1
        end;
    write sum
end.
```

図 10: 作成実験 2 の原始プログラム例

いう意味だけになる。write文の意味は,「write文の後ろに続く変数の値をスペース1個で区切りながら1行に表示し，最後に改行コードを出力する」である。また，変数を使用する場合は，構文図の outblock の中で宣言しておかなければならないとする。

3.3.2 変数の処理

変数は原始プログラム中で数値を記憶するために使用される。このため，目的プログラムでは，原始プログラムでの変数に相当する記憶場所を目的計算機内のメモリに確保し，そこを使用しなければならない。機械語では記憶場所はメモリのアドレスで管理されるため，それぞれの変数がどのアドレスに対応するかをコンパイラが決定し，その決定に従って，目的プログラムを生成する必要がある。この変数とアドレスの対応を表現する表を，記号表と呼ぶ。

構文図と意味論によれば，使用する変数を必ず原始プログラムの最初の段階で宣言しなければならないため，コンパイラにとっては都合がよい。すなわち，原始プログラムを読み込む最初の段階 (outblock) で，そのプログラムが使用する全変数を知ることができるため，この段階で変数表を作成できる。プログラム本体 (statement) のコンパイルでは，この記号表を参照しながら，変数を機械語のアドレスに変換すればよい。

実際の変数の記憶場所として，目的計算機のデータ領域を使用する。目的計算機の命令体系によれば，type-4-1 のレジスタ – メモリ間の演算・転送命令を実行するとき，メモリアドレスに自動的にデータセグメントレジスタの値が加算される。このため，データ領域は 0 番地から始まっていると考えることができる。また，データセグメントレジスタの値は十分大きな値が設定されているため，機械語プログラムの格納されているプログラム領域とは，独立であると仮定してよい。

結局，記号表を作る際のアドレスの決定は，最初に outblock で変数が宣言される際の順番に従って，0,1,2,... と付けていけばよい。記号表は変数とアドレスのペアのリストとなる。例えば，図 12 で表されるような構造となる。原始プログラムに変数が現れたら，この記号表の変数名の部分を探索して，一致する変数名に対応するアドレスを記憶場所として使用する。変数の使われ方は，代入文の代入記号「:=」の左右で異なる。左にある場合は，右で計算された結果をその変数に対応するアドレスへ格納 (store) することになり，右にある場合は，その変数に対応するアドレスの内容を計算に使用するため，load命令かあるいはレジスタ – メモリ間の演算命令が使用される。

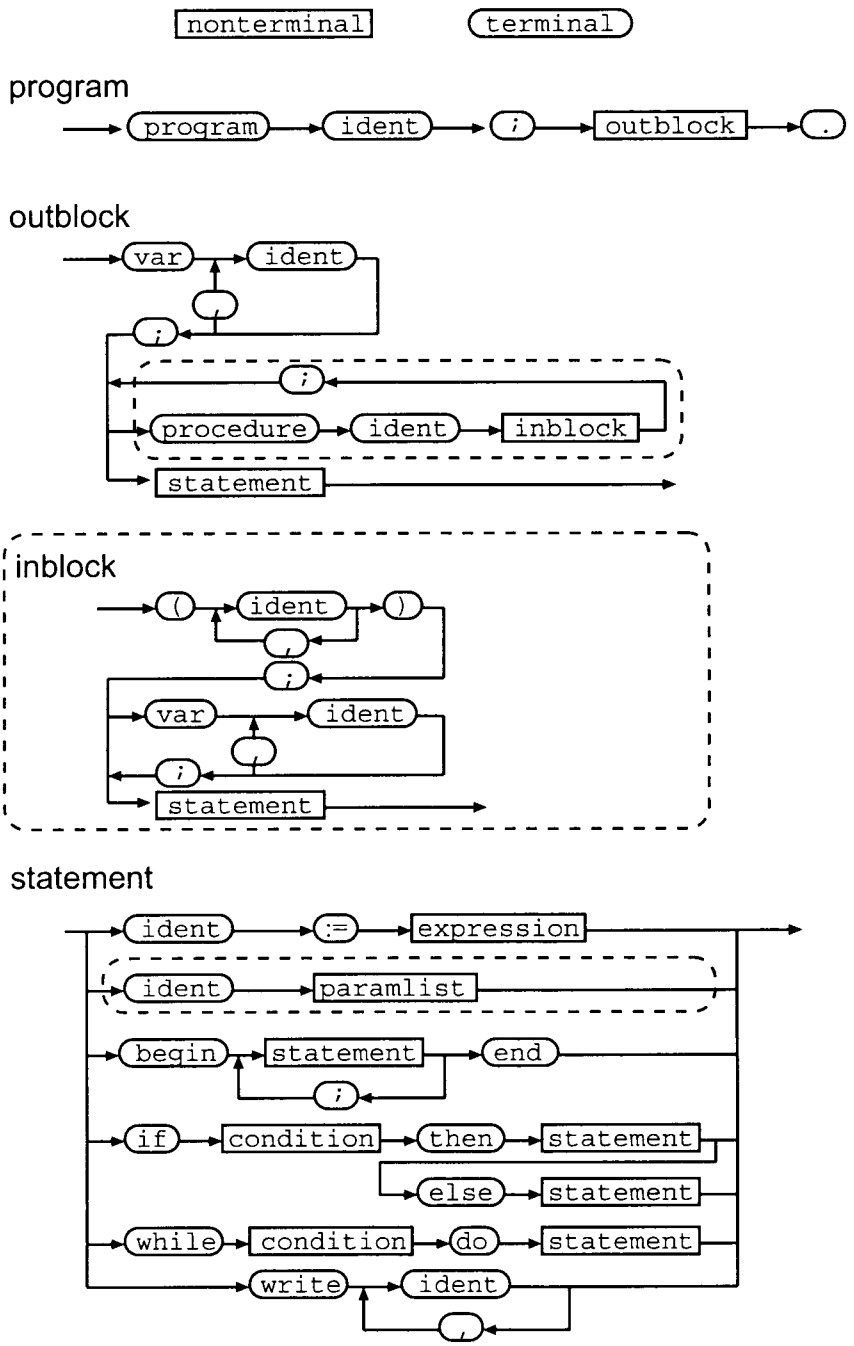


図 11: 作成実験 2, 3, 4 の構文図 (その 1)

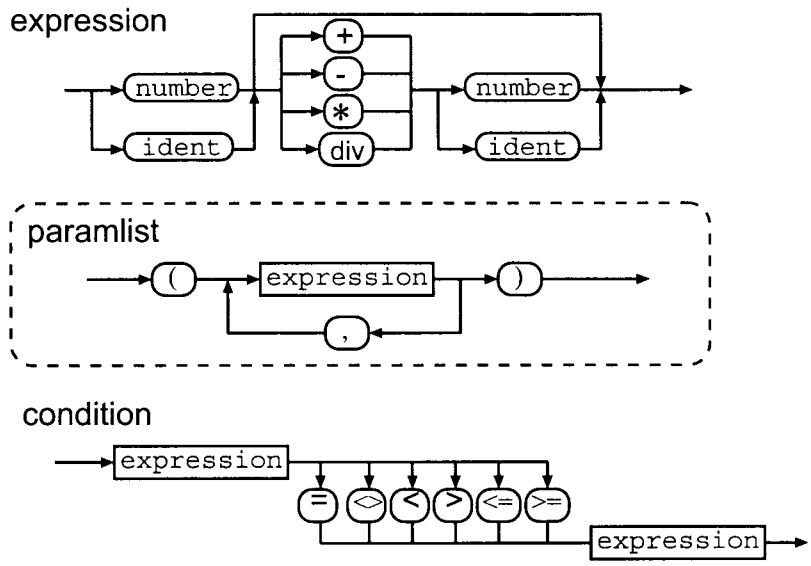


図 11: 作成実験 2, 3, 4 の構文図 (その 2)

変数名	アドレス
x	0
y	1
z	2
i	3
sum	4
⋮	⋮

図 12: 記号表の例

3.3.3 if 文, while 文

if 文と while 文に対応する目的プログラムは，図 13 のような構成となる。if 文の then 部，else 部，while 文の本体はすべて statement であり，statement をコンパイルするルーチンにまかせればよい。条件文のコンパイルと，その条件によってどこへジャンプするかを目的プログラムに反映させることが，主な作業である。

条件文は 2 つの expression と大小を表す記号からなっている。expression の値を計算する目的プログラムは expression をコンパイルするルーチンがやってくれる。注意するのは，目的プログラム中で 1 つ目の expression を計算した結果をどこかにしまっておかなければならないことである。例えば，expression を計算するプログラムで使用するレジスタに 1 つ目の expression の計算結果を保存していれば，2 つ目の expression を計算する部分で 1 つ目の結果を破壊してしまう。1 つ目の expression を計算した結果は，計算に使用しないレジスタ，あるいはメモリ上に格納するようにすればよい。

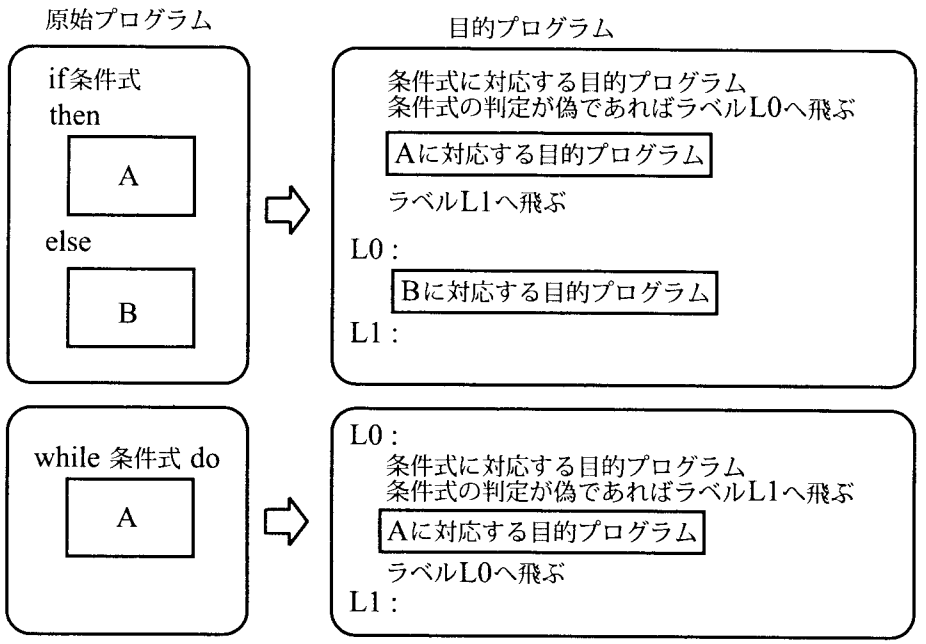


図 13: if 文と while 文の目的プログラムの構成

3.4 作成実験 3

3.4.1 原始プログラムの構文と意味論

作成実験 3 では，数式 (expression) を一般的な式へ拡張する。原始プログラムの例を図 14 に，作成実験 2 と異なる構文図の部分 (expression のみ) を図 15 に，それぞれ示す。作成実験 1,2 では，単なる加減乗除を 2 つの数値または変数に対して行うことができるのみであったが，作成実験 3 では加減乗除の組み合わせや括弧の使用を許した数式を扱う。当然，乗除の演算が加減の演算よりも優先され，さらに括弧の中の計算が優先される。計算機の機械語では一般に，加減乗除を 1 つの命令で 1 種類しか行えないので，構造を持った数式のどの部分とどの部分をどのような順序で計算すればよいかを，正確に解析しなければならない。

本作成実験では，数式の解析を行う 2 種類の方法を作成する。1 つは演算子順位構文解析法と呼ばれるもので，もう 1 つは expression 以外の部分の解析で使用している構文図をそのまま C 言語の関数で実現する方法である。概念的には，前者はボトムアップ構文解析，後者はトップダウン構文解析と呼ばれる解析法の仲間 (前者はやや特殊) である。

また，数式を処理する場合はやや複雑な処理を行うため，直接アセンブラ言語にコンパイルするのではなく，中間的な言語にまず変換してから，最後にまとめてアセンブラ言語に変換する手法が使われる。中間的な言語で書かれたプログラムを中間コードと呼び，本実験では「4 つ組」というものを利用する。以下，演算子順位構文解析法とトップダウン構文解析法について簡単に説明する。

3.4.2 演算子順位構文解析

数式のどの部分から先に計算すればよいかは，数式の構造を書けばよく分かる。図 16 は数式の構造の例である。数式全体の値は各部分構造の結果を演算して得られるため，構造の内側のものから順番に計算していけばよい。数式を目的プログラムに変換するためには，この構造を解析し，一番内側のものから順番に目的コードを出力する。

出力される目的プログラムは，部分構造ごとに計算結果がメモリ上のある場所に格納され，その部分構造は変数と同じように扱えばよい。すなわち，ある部分構造の結果を使って (優先順位が

例 1:

```
program pi;
var i, rand, ca, cm, pi, qpi, x, y;
begin
    i := 0;
    qpi := 0;
    rand := 3827;
    ca := 8358;
    cm := 3797;
    while i < 1000 do begin
        rand := (rand + ca) * cm - (((rand + ca) * cm) div 100000) * 100000;
        rand := rand div 10;
        x := rand div 10;
        rand := (rand + ca) * cm - (((rand + ca) * cm) div 100000) * 100000;
        rand := rand div 10;
        y := rand div 10;
        if x * x + y * y < 1000000 then
            qpi := qpi + 1;
        i := i + 1
    end;
    pi := qpi * 4;
    write pi
end.
```

図 14: 作成実験 3 の原始プログラム例

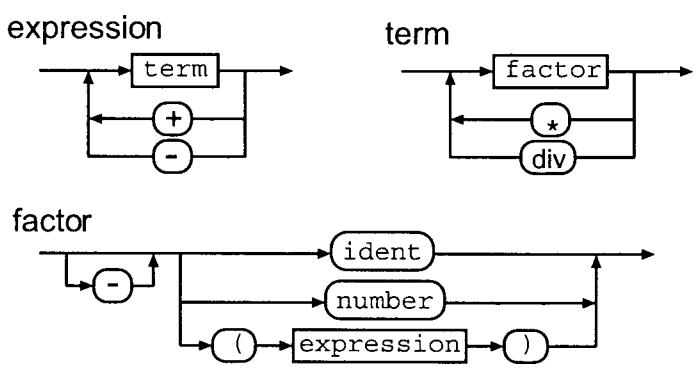


図 15: 作成実験 3 の構文図 (変更分)

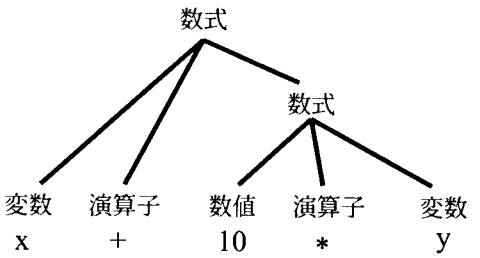


図 16: 数式の構造

低い) より外側の部分が計算される場合は，内側の部分を変数と見なしてその結果が格納されているアドレスを用いて計算するような，プログラムを出力する。

一つの部分構造を計算するプログラムを「4つ組」と呼ばれる中間コードで表現し，先に計算するべきものから順に並べたものが図 17 である。それぞれの 4 つ組は，4 つの記号の組から成っている。4 つ組を ($a1$，$a2$，$a3$，$a4$) で表せば，$a1$ が演算子，$a2, a3$ が被演算子，$a4$ が結果の格納場所を表す。例えば，(*，x，10，T1) という 4 つ組は，x という変数と 10 という数値をかけて，T1 という場所に格納するということを意味している。格納場所は数式には明示的に書いてないため，コンパイラが決定しなければならない。各部分構造に 1 つづつ異なる格納場所を割り当てれば容易に決定できる。

構文の解析は若干複雑なため，このような中間コードを仮定することにより，全体の処理を見通しのよいものにできる。以下の説明では 4 つ組の出力を最終目的とし，4 つ組から実際の目的プログラムへの変換については述べない。

構造を解析するために数式を左から右に見ていき，スタックに今すぐに使わない演算子，被演算子を積み，スタックに積む・降ろすを演算子の優先順位で決定していく解析手法を，演算子順位構文解析と呼ぶ。スタックとは，要素を 1 度に 1 つづつ入れたり・出したりできるデータ構造で，入れた順番と逆順に (すなわち，最初に入れたものが最後に) 出てくるものである。数式処理における優先順位の低い演算子は最初に出てきても後で使われるため，このデータ構造は非常に都合がよい。

直感的には，数式のある部分式に対応するプログラムを出力するタイミングは，その部分式が右隣の演算子の優先順位より高い場合である。もし，右隣の演算子の方が優先順位が高い場合は，その部分式に対応するプログラムは，右隣のより優先順位の高い部分式の計算プログラムを出力した後になる。このため，右隣の優先順位が高い場合にはどんどんスタックに入れていき，右隣の処理が済み，右隣の優先順位が低くなった時点でスタックから取り出し，その部分の計算プログラムを出力する。例えば，次のような数式の処理を考える。ここで，最初と最後の「$」は数式の始まりと終わりを表す記号である。

```
$ x + 10 * y $
```

この式を左から右に見ていき，10 のところまで来た時点では，「x + 10」の 4 つ組を出力できない。なぜなら，その後にどのような演算子が来るか分からないからである。次に「+」や「−」のように左の演算子よりも優先順位が同じか低いものが来た場合は出力できる。この場合は次に「*」

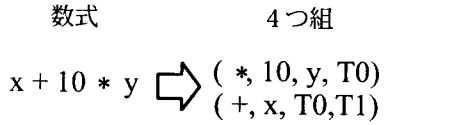


図 17: 4 つ組の例

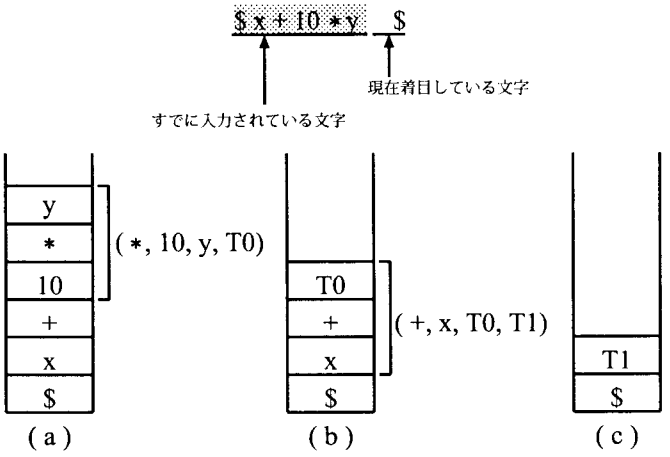


図 18: 演算子順位法のスタックの変化

が来るので出力できない。出力されないので，すべての項がスタックに積まれている。さらに y が来るが，これは演算子ではないので，やはりそのままスタックに積まれる。ここまでの様子を図 18(a) に示す。最後に数式の終わりを表す「$」が来てはじめて，スタックから順番に部分式が取り出され，4 つ組が出力される。最初に「10 * y」の部分が取り出され，4 つ組が出力される。1 つ目の 4 つ組が取り出された後の様子を図 18(b) に示す。上で述べたように，数式の部分式を処理した後，その部分式は変数と同じように扱われる。したがって，部分式の格納場所である番地 T1 が，取り出された部分式の代わりにスタックのトップに置かれている。さらに，次の部分式を処理した結果を図 18(c) に示す。スタックには数式の最初を表す「$」と全体の計算結果が格納されているアドレスのみが入っており，現在見ている数式の部分が数式の終わりを表す「$」であることから，処理が終了したことが分かる。結果として，正しい 4 つ組が得られている。

ここまでの説明は非常に直感的なものである。詳しいアルゴリズムと各演算子の優先順位の決め方を付録 D につけるので参照せよ。

3.4.3 トップダウン構文解析

作成実験 1, 2 で，構文図の program や statement などの部分を処理した手法は，トップダウン構文解析手法の 1 つである。そこでは，トークンを 1 つ読んでは構文図のどこに行くかを決定し，次の構文が他の構文を呼んでいる場合は，次の構文を処理するためのルーチンを呼ぶことによって解析した。数式も program や statement の部分と同じような構文図 (図 15) で表現できるため，同じ手法によって処理できる。

足し算・引き算を表現する expression 構文の中で，再帰的にかけ算・わり算を表現する term が呼ばれている。各構文を C 言語の関数で表現すれば，expression を処理する関数が最後に実際の目的プログラムを出力する場合，すでに term の処理は終わっている。なぜなら，term の処理が終わらなければ，term を呼び出した expression へ実行は戻ってこないからである。こうして，常にかけ算・わり算を計算する目的プログラムが，足し算・引き算を計算する目的プログラムよりも先に出力される。C 言語の関数は再帰呼び出しができるために，構文が再帰的に書かれている場合は大丈夫である。これが直感的な仕組みである。

このような方法で作成すれば，演算子順位法と比べて非常に簡単に，数式をコンパイルするルーチンを作成することができる。簡単に作成できる理由は，C 言語の再帰呼び出しにある。再帰呼び出しには必ずスタックが使われているが，このスタックは C 言語が自動的に用意してくれるものである。この自動的に用意されたスタックを，この手法はうまく利用している。演算子順位法ではスタックをみずから作って操作していたために，複雑になったと考えることができる。

例 1:

```
program hanoi;
var n;

procedure hanoi(a, b, c, n);
var i;
if n = 1 then write a, b
else
    begin
        hanoi(a, c, b, n-1);
        write a, b;
        hanoi(c, b, a, n-1)
    end;

begin
    n := 5;
    hanoi(1, 2, 3, n)
end.
```

図 19: 作成実験 4 の原始プログラム例

3.5 作成実験 4

作成実験 4 では再帰呼び出し可能な手続きを含む言語を扱えるように作成実験 3 のコンパイラを拡張する。

3.5.1 原始プログラムの構文と意味論

図 11 の expression を図 15 のもので置き換え，さらに点線の内部も含むものが，作成実験 4 で作成するコンパイラの原始プログラム言語の構文である。図 19 に原始プログラムの例を示す。拡張は手続きが導入され，手続き定義，手続き呼び出しが行えることである。手続きは再帰呼び出し可能とし，手続きの中で手続きは定義できないものとする。引数は値渡しであり，手続きの返り値はないので，手続きの処理結果は大域変数を介して行われる。

再帰呼び出し可能とするためには，目的プログラムがどのように実行されるかをよく検討する必要がある。目的プログラムが実行されるときのモデルを実行時環境と呼ぶ。次の節では，再帰呼び出しを可能とする一つの実行時環境を説明する。

3.5.2 実行時環境

プログラムの実行時の振る舞いは動的であり，原始プログラムの手続きに現れる一つの局所変数が，実行時には手続きの再帰呼び出しによって，複数の実体に対応する場合がある。これを処理するために，手続きが呼ばれるたびに新しい変数領域を局所変数のために確保する必要がある。その他，手続きは呼ばれるときに引数を与えなければならないし，手続きが終了した後に呼ばれた場所に戻るために戻り番地を記憶しなければならない。これらの手続き呼び出し・実行に必要な記憶場所として，呼び出しのたびにスタック上に連続した領域を確保していく手法が，再帰手続きの実行時環境としては一般的である。この連続した領域をフレームと呼ぶ。手続きが終了した場合，確保された領域はスタック上から削除される。この順序は呼ばれた順番と常に逆であるため，スタックの利用が適している。

手続きの呼び出しのたびに局所変数の格納場所が異なるということは，コンパイル時に変数のアドレスが決定できないことを意味する。しかし，一回の手続き呼び出しの際にスタック上に確保される領域は同じ大きさであるため，その領域のある点からの相対的なアドレスは一定であ

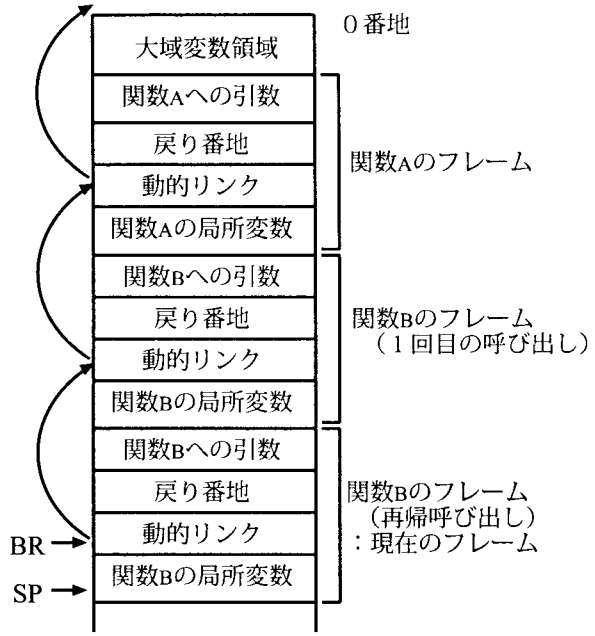


図 20: 実行時環境

る。これを利用して，手続きを実行するために確保される領域の相対アドレスで局所変数をアクセスするような目的プログラムを生成すればよい。実行時には，手続きの呼び出しごとにベースレジスタが新しく確保された領域の決まった場所を指し，そのベースレジスタからの相対アドレスで変数をアクセスする。このために，ベースレジスタ相対でアドレスを決定する命令が，それぞれの演算命令，転送命令に存在する。

局所変数に対して，大域変数は複数の実体に対応することはないため，ある決まった領域を確保しておけばよい。

ここまで説明してきた実行時環境をまとめたものを図 20 に示す。この図は手続きがネストして 3 回呼ばれているところであり，最後の 2 つのフレームは再帰呼び出しによる同じ手続きのものである。スタックは 0 番地を底として大きい番地に延びていくので，メモリのイメージに従い一番上がスタックの底である。スタックの底には，大域変数の領域が確保されている。一つのフレームは，引数，戻り番地，動的リンク，局所変数の領域からなる。引数は呼び出し側が積み，call 命令によって戻り番地が自動的に積まれる。動的リンクは，手続きが終了して戻る場合にベースレジスタを元の値に戻すために使われる。

3.5.3 記号表

手続きを導入したことによって，記号表の扱いも異なるものとなる。手続きの名前からその手続きのプログラムのアドレスを参照するために，変数と同じ記号表を用いる。変数の格納場所が手続きの開始アドレスに変わっただけである。また，変数に関しては大域変数と局所変数の区別を行う必要がある。手続き内の変数は，局所変数としてその手続きで定義されているか，または引数である場合は局所変数とみなされるが，そうでない場合は大域変数とみなされる。コンパイラでも 2 つの記号表を用意し，1 つを大域変数用，もう 1 つを局所変数用の記号表とする。手続きのコンパイルでは，変数のアドレスを求めるために，まず局所変数用の記号表を参照し，そこにない場合は大域変数用の記号表を参照すればよい。

手続きの開始アドレスは大域変数用の記号表に入れなければならない。なぜなら，手続きの名前は原始プログラム全体で参照できるからである。

3.5.4 forward文

図 21(a) のような原始プログラムをコンパイラが先頭から順番に処理する場合，proc1 の目的プログラムが最初に出力される。しかし，proc1 を処理している段階では，まだ proc2 を処理していないために，proc1 の中で呼ばれている proc2 の開始アドレスが決定できない。また，コンパイラは proc2 があることも知らないので，エラーとしてもよい。このために，前方参照をする名前は原始プログラムの先頭で，使うことを宣言しておく必要がある。例えば，forward 文を導入し，図 21(b) のように先頭で宣言するようにすれば，コンパイルの初期段階で前方参照をする全手続きを知ることができる。コンパイラは forward 文を読み込んだ段階で，開始アドレスを決定し，実際の本体にそのアドレスを使用すればよい。forward 文を使用せずとも，まだ宣言されていない手続きが呼び出されていることを発見した段階で，開始アドレスを仮定してもよい。

```
program forwardref;
var i;
        ⋮
procedure proc1(a);
var j;
begin
    ⋮
    proc2(j)
    ⋮
end;

procedure proc2(a);
var k;
        ⋮
begin
    ⋮
end.
```

(a)

```
program forwardref;
var i;
        ⋮
forward proc2;
procedure proc1(a);
var j;
begin
    ⋮
    proc2(j)
    ⋮
end;

procedure proc2(a);
var k;
        ⋮
begin
⋮
end.
```

(b)

図 21: 手続きの前方参照の例

4 実験報告書

実験の報告書には以下の内容を必ず含むこと。

(1) 最終的に作成した記号表の構造と操作関数の仕様

(2) if 文，while 文のコンパイルの仕様

(3) 作成した演算子順位構文解析の仕様

(4) 作成したトップダウン構文解析による数式処理の仕様

(5) 演算子順位構文解析とトップダウン構文解析の比較検討

(6) 手続きの実現の仕様

(7) エラー処理について

(8) 全体の構成概念図，実現した各関数の依存関係図

(9) 作成したコンパイラが正しく動いているという証拠

(10) こちらから指定したプログラムの実行時間

(11) その他，工夫した点について

(12) コンパイラのソースプログラムの全リストにコメントをつけたもの

5 参考文献

中田育男：「コンパイラ」産業図書 1991, 2900 円 (初心者用だが，実行時環境の説明がない)
佐々政孝：「プログラミング言語処理系」岩波書店 1989, 4600 円 (初心者用)
エイホ，ウルマン，セシィ (原田賢一訳)：「コンパイラ I · II」サイエンス社 1991,
各巻 5768 円 (本格的なコンパイラのテキスト)

付録A　目的計算機のアーキテクチャ

A.1　概要

以下のようなレジスタを持つレジスタマシンである。図22に構成を示す。

汎用レジスタ：　R0, R1, R2, R3
スタックポインタ：　SP(R4)
ベースレジスタ：　BR(R5)

プログラムカウンタ：　PC
コンディションレジスタ：　CR
プログラムセグメントレジスタ：　PS
データセグメントレジスタ：　DS

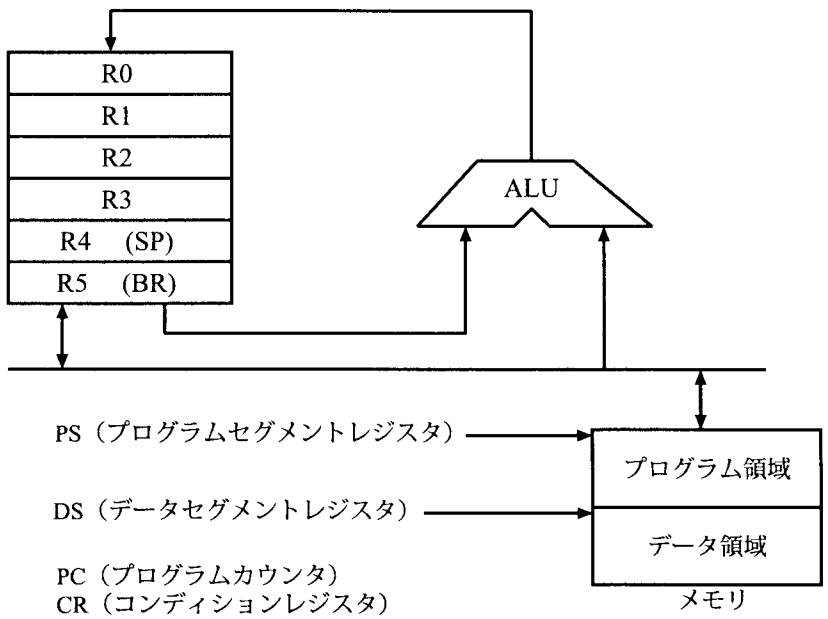


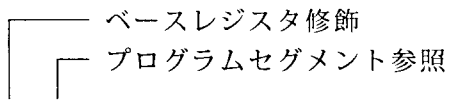
図 22: 目的計算機の構成

各レジスタの大きさは1word(= 32bit) である。プログラムの実行開始時のレジスタの初期値は次の通りである。

R0,R1,R2,R3 == 不定
SP == −1
BR == 0
PC == 0
CR == 0
PS == 0
DS == 十分大きな値

A.2　命令の語長

各命令の語長は一定で，1word である。命令の種類は次のようなものがある。



type1:	opcode	–	–	–	–	–
type2:	opcode	–	–	register	–	–
type3:	opcode	–	–	register	register	–
type4-1:	opcode	–	–	register	–	addr
type4-2:	opcode	1	–	register	–	offset
type4-3:	opcode	–	1	register	–	addr
type5:	opcode	–	–	register	–	immed
type6:	opcode	–	–	–	–	addr

A.3 アドレッシング

type4 と type6 の命令はメモリを参照する。それぞれのタイプの実効アドレスの計算は，以下のように行う。

type4-1: DS + addr
type4-2: DS + BR + offset
type4-3: PS + addr
type6: PS + addr

ただし $0 \leq \text{addr} \leq 65535$, $-32768 \leq \text{offset} \leq 32767$ である。
また，実行される命令のアドレスは次の計算によって決められる。

次に実行する命令のアドレス：PS + PC

A.4 命令体系

Rx, Ry :レジスタオペランド (R0, R1, R2, R3, R4, R5, BR, SP)
CR :コンディションレジスタ
(label) :label が指すアドレス
immed :即値 ($-32768 \leq \text{immed} \leq 32767$)
[x] :アドレス x のメモリの内容 (値)
[addr],[BR+offset],[(label)] に関しては，各命令タイプの実行アドレスの計算法に従う。

命令(タイプ)	アセンブラ表記		動作	実行サイクル (1サイクル 1μsec.)
Load (type4-1)	`load`	Rx, addr	Rx ⇐[addr];	5
Load (type4-2)	`load`	Rx, offset(`BR`)	Rx ⇐[`BR`+offset];	7
Load (type4-3)	`load`	Rx, label	Rx ⇐[(label)];	5
Load register(type3)	`loadr`	Rx, Ry	Rx ⇐ Ry;	1
Load immediate(type5)	`loadi`	Rx, immed	Rx ⇐ immed;	3
Store (type4-1)	`store`	Rx, addr	[addr] ⇐ Rx;	5
Store (type4-2)	`store`	Rx, offset(`BR`)	[`BR`+offset] ⇐ Rx;	7
Store (type4-3)	`store`	Rx, label	[(label)] ⇐ Rx;	5
Add (type4-1)	`add`	Rx, addr	Rx ⇐ Rx + [addr];	6
Add (type4-2)	`add`	Rx, offset(`BR`)	Rx ⇐ Rx + [`BR`+offset];	8
Add (type4-3)	`add`	Rx, label	Rx ⇐ Rx + [(label)];	6
Add register(type3)	`addr`	Rx, Ry	Rx ⇐ Rx + Ry;	2
Add immediate(type5)	`addi`	Rx, immed	Rx ⇐ Rx + immed;	4
Subtract (type4-1)	`sub`	Rx, addr	Rx ⇐ Rx − [addr];	6
Subtract (type4-2)	`sub`	Rx, offset(`BR`)	Rx ⇐ Rx − [`BR`+offset];	8
Subtract (type4-3)	`sub`	Rx, label	Rx ⇐ Rx − [(label)];	6
Subtract register(type3)	`subr`	Rx, Ry	Rx ⇐ Rx − Ry;	2
Subtract immediate(type5)	`subi`	Rx, immed	Rx ⇐ Rx − immed;	4
Multiply (type4-1)	`mul`	Rx, addr	Rx ⇐ Rx * [addr];	10
Multiply (type4-2)	`mul`	Rx, offset(`BR`)	Rx ⇐ Rx * [`BR`+offset];	12
Multiply (type4-3)	`mul`	Rx, label	Rx ⇐ Rx * [(label)];	10
Multiply register(type3)	`mulr`	Rx, Ry	Rx ⇐ Rx * Ry;	6
Multiply immediate(type5)	`muli`	Rx, immed	Rx ⇐ Rx * immed;	8
Divide (type4-1)	`div`	Rx, addr	Rx ⇐ Rx / [addr];	10
Divide (type4-2)	`div`	Rx, offset(`BR`)	Rx ⇐ Rx / [`BR`+offset];	12
Divide (type4-3)	`div`	Rx, label	Rx ⇐ Rx / [(label)];	10
Divide register(type3)	`divr`	Rx, Ry	Rx ⇐ Rx / Ry;	6
Divide immediate(type5)	`divi`	Rx, immed	Rx ⇐ Rx / immed;	8
Compare (type4-1)	`cmp`	Rx, addr	`CR` ⇐ Rx − [addr];	6
Compare (type4-2)	`cmp`	Rx, offset(`BR`)	`CR` ⇐ Rx − [`BR`+offset];	8
Compare (type4-3)	`cmp`	Rx, label	`CR` ⇐ Rx − [(label)];	6

Compare register(type3)	`cmpr`	Rx, Ry	CR ⇐ Rx − Ry;	2
Compare immediate(type5)	`cmpi`	Rx, immed	CR ⇐ Rx − immed;	4
Jump (type6)	`jmp`	label	PC ⇐ (label);	12
Jump Non Zero(type6)	`jnz`	label	CR ≠ 0 ならば PC ⇐ (label);	12
Jump Zero(type6)	`jz`	label	CR = 0 ならば PC ⇐ (label);	12
Jump Greater Than(type6)	`jgt`	label	CR > 0 ならば PC ⇐ (label);	12
Jump Greater Equal(type6)	`jge`	label	CR ≥ 0 ならば PC ⇐ (label);	12
Jump Less Than(type6)	`jlt`	label	CR < 0 ならば PC ⇐ (label);	12
Jump Less Equal(type6)	`jle`	label	CR ≤ 0 ならば PC ⇐ (label);	12
Call(type6)	`call`	label	SP ⇐ SP + 1; [SP] ⇐ PC; PC ⇐ (label);	25
Return(type1)	`return`		PC ⇐ [SP]; SP ⇐ SP − 1;	15
Push(type2)	`push`	Rx	SP ⇐ SP + 1; [SP] ⇐ Rx;	6
Pop(type2)	`pop`	Rx	Rx ⇐ [SP]; SP ⇐ SP − 1;	6
Read decimal(type2)	`read`	Rx	Rx ⇐ キーボードから読み込んだ数;	0
Write Character(type2)	`writec`	Rx	Rx の値を文字として出力;	0
Write Decimal(type2)	`writed`	Rx	Rx の値を数字として 10 進数で出力;	0
Halt(type1)	`halt`		マシンの停止;	0

A.5 目的計算機シミュレータの使い方

目的計算機シミュレータのコマンドは次のような形式をしている。

```
% sr  input-file
```

input-file は機械語のファイルである。付録Bのアセンブラの出力が使われる。実行の最後に実行にかかった時間が表示される (単位はマイクロ秒)。

付録B　アセンブラ

B.1　アセンブラの1行のフォーマット

type1: [label:]opcode
type2: [label:]opcode register
type3: [label:]opcode register register
type4:
  type4-1: [label:]opcode register addr
  type4-2: [label:]opcode register offset(BR)
  type4-3: [label:]opcode register label
type5: [label:]opcode register immed
type6: [label:]opcode label
data: [label:]`data` value

各項は，スペース，あるいはコンマ(",")で区切られなければならない。

label == 任意の識別子
　　英字で始まり英数字が続くもの(英字には $, _, . を含む)
　　例: `L01`, `loop`, `$111` など
opcode == 命令のニーモニック
　　例: `load`, `loadr`, `addi` など(付録Aの命令体系のニーモニックを使う)
register == `R0`, `R1`, `R2`, `R3`, `R4`, `R5`, `SP`, `BR` (ただし，`R4` = `SP`, `R5` = `BR`)
　　'R' は小文字でもよい。
addr, offset == 数字(データセグメントのアドレス)
　　DS + addr, または DS + BR + offset を計算して実効アドレスが決定される。
　　例: `0`, `1`, `10`, `22` など
immed == 数字あるいは文字。文字はクオーテーションに囲まれたもの。
　　例: `333`, `10`, `'a'`, `'A'`, `'\n'`(改行),
　　' '(スペース),
　　'\''(シングルクオート),
　　'\\'(バックスラッシュ) など
`data` == 疑似命令。value を 1word のデータに変換したものに置換される。
　　value は addr, offset, label あるいは immed (ただし $-2^{31} \leq \text{immed} \leq 2^{31} - 1$)

B.2　アセンブラの使い方

アセンブラコマンドは次のような形式である。

`%` `asm` input-file [output-file]

input-file はアセンブリ言語のプログラムである。このファイルをアセンブルして output-file という名前の機械語ファイルを出力する。output-file は省略可能であり，省略された場合は標準出力に機械語が出力される。

付録C　字句解析ルーチンgetsymの仕様

ソースファイルは単なる文字の列である。このファイルからコンパイルするための最初の処理は字句解析と呼ばれ，ソース言語の最小の単位を見つける処理を行う。例えば，“100” という3文字は，その文字列の両側が空白の場合は数字の100を表すが，前にアルファベットがあれば変数などの記号 (例えば “a100” など) となる。このような違いをきちんと見分けて，ソースファイルの頭から順に，部分的な文字列をある意味のかたまり (記号であるとか数字であるなど) として切り出していくのが字句解析である。

この字句解析ルーチンに関しては，あらかじめこちらで作成した字句解析処理関数を使用してよい。この関数の仕様を以下に示す。

関数名： `getsym()`　呼び出されるごとにソースファイルから1つの意味ある部分文字列とその意味を返す。

関数への入力：

引数: なし

global 変数: `FILE` 型の `infile` という global 変数に，入力ファイルのストリームポインタ (`fopen()` の返り値) を入れておくこと。

返り値： 関数は値を返さないが，global 変数 `tok` に1つの単位を切り出した結果が入っている。`tok` は `TOKEN` 型の変数であること。`TOKEN` 型の宣言は次の通りである。

```
typedef struct {
    int attr;
    int value;
    char charvalue[MAXIDLEN + 1];
    int sline;
} TOKEN;
```

ただし `MAXIDLEN` は識別子の最大文字数を表すマクロ定数である。`sline` にはそのトークンがあった原始プログラムの行が入っている (エラーメッセージを表示するとき利用できる)。`attr` には切り出された文字列の種類として，次のようなマクロ定数値が入る。

IDENTIFIER：変数，プログラム，関数の名前にあたる文字列を表す。そのときの文字列は `charvalue` に入っている。

NUMBER：文字列が数字を表しているときこの値が入る。また，実際に数値に変換された値は `value` に入っている。

RWORD：文字列が予約語 (`begin`, `end`, `div`, `var` など) の時にこの値が入る。また，それぞれの予約語に対応する以下のマクロ定数値が `value` に入っている。

```
#define BEGIN        260
#define DIV          261
#define DO           262
#define ELSE         263
#define END          264
#define IF           265
#define PROCEDURE    266
#define PROGRAM      267
#define THEN         268
#define VAR          269
#define WHILE        270
#define READ         271
#define WRITE        272
```

SYMBOL : 文字列が記号 (+,*,<, :=, <=など) の場合，この値が入る。また，それぞれの記号に対応する以下の値が value に入っている。

```
#define PLUS          '+'
#define MINUS         '-'
#define TIMES         '*'
#define LPAREN        '('
#define RPAREN        ')'
#define EQL           '='
#define COMMA         ','
#define PERIOD        '.'
#define SEMICOLON     ';'
#define BECOMES       256       /* := */
#define LESSTHAN      '<'
#define LESSEQL       257       /* <= */
#define NOTEQL        258       /* <> */
#define GRTRTHAN      '>'
#define GRTREQL       259       /* >= */
#define COLON         ':'
```

ENDFILE : 入力ファイルの最後に到達したことを表す。

SYMERR : 認識できないトークンが現れたことを表す。

TOKEN の定義と上記の各 define 文のヘッダファイルがあるので，それを利用すること。

使い方 : tok 変数は getsym のプログラムの中で extern 宣言されているので，ユーザで用意すること (単に global 変数 tok を TOKEN 型で宣言すればよい)。また getsym() を使う前に 1 度だけ init_getsym() 関数 (引数なし) を呼んで，getsym() の初期化をしなければならない。プログラムはライブラリとなっているので，次のようにすれば自動的にリンクされる。

% cc *user-program.c* ... -lics -L*lpath* -I*ipath*

「-lics」は libics.a をライブラリとして参照するオプションである (libics.a は getsym.c から生成した情報系実験用ライブラリ)。ライブラリ検索パスを指定するオプション-L に，libics.a が存在するディレクトリのパス *lpath* を与える必要がある。

また，tok 変数の中の値の意味を表すヘッダファイルとして getsym.h がある。getsym を利用するときは，このヘッダファイルを include して使うこと。この中に TOKEN 型の宣言と値の定義が入っている。getsym.h が存在するディレクトリのパス *ipath* も，インクルードファイル検索パスを指定するオプション-I に与える必要がある。

付録D　演算子順位法のアルゴリズム

D.1　アルゴリズム

"$" を式の終わりを表す記号とする。また，各演算子の順位関係 ($\ll$) を表す順位関数の作り方は次のD.2で説明する。

入力：$a_1 a_2 a_3 \ldots a_m$ $
処理：
初期設定：スタックに "$" を積む
以下を繰り返す

1) 次の入力記号 a_j を読む
2) a_j が演算数 (演算子でない) ならば a_j をスタックに積む
3) a_j が演算子ならばスタックの最も上の演算子 a_i との関係によって次のいずれかの処理をする
   A) $a_i \gg a_j$ である間，以下を繰り返す
      - a_i と演算数をスタックから降ろして出力し，結果の演算数 (格納場所) をスタックに積む
   B) $a_i \ll a_j$ なら a_j をスタックに積む
   C) $a_i = a_j$ のとき
      - a_i が "(" なら a_i をスタックから降ろす (その分最上部の演算数をずらす)
      - a_i が "$" なら終了

D.2　順位関数

$a \ll b$を表現するためにテーブルを作る方法もあるが，ここでは，作り方が簡単な順位関数を用いる方法を説明する。順位関数は2つの関数 f, g からなり，それぞれ記号を引数とし，数値を返す関数である。f と g が返す数値を元に次のように記号 a, b の順位は決定される。

$$
\begin{aligned}
f(a) > g(b) \quad &\text{ならば} \quad a \gg b \\
f(a) = g(b) \quad &\text{ならば} \quad a = b \\
f(a) < g(b) \quad &\text{ならば} \quad a \ll b
\end{aligned}
$$

f, g の値は次のようにして決める。

2項演算子に対する優先順位が n 段階あるとする。そのときの順位関数は，各優先順位 $i (i > 0)$ に $2i-1$ と $2i$ の2つの数を割り当てる。そして，その演算子 op が左結合的ならば $f(op) = 2i$, $g(op) = 2i-1$ とし，右結合的ならば逆の値とする。括弧やi (演算数) や "$" に対する順位関数は，適当な $m > 2n$ について，$f(\text{")"}) = f(\text{i}) = m+1$, $g(\text{"("}) = g(\text{i}) = m$, その他は0とする。

この決め方は次のような考え方による。括弧の優先順位はその内部にあるものより小さく，その外部にあるものよりも大きい。iは "()" と同じと見なす。"()" 全体やiは左から先に演算する。左端の "$" は "(" と同じ，右端の "$" は ")" と同じである。

単項演算子 uop については，2項演算子と異なる記号の時は $g(uop)$ をどの $f(op)$ の値よりも大きな値，$f(uop)$ を op の優先順位から決まる値にすればよい。"-" のように，2項演算子と同じ記号が単項演算子として使われるときは，やや面倒である。しかし，"–" が単項演算子として使われる場所は "$", "(", "*", "div" の直後だけであるので，そこに出てきた "–" を異なる記号として

置き換えて使えばよい。例えば，本実験の場合は，次のようになる。ここで，“!”は単項演算子の“–”である。

	+	-	*	div	!	(	)	i	$
f	2	2	4	4	6	0	11	11	0
g	1	1	3	3	15	10	0	10	0

付録E　作成実験１の簡単な構文解析プログラム例

- メイン関数ファイル

```
#include <stdio.h>
#include <strings.h>
#include <getsym.h>
TOKEN tok;
FILE *infile,*outfile;

void compiler(void);

int main(int argc, char *argv[]){
    fprintf(stderr,"Simple compiler: compile start.\n");
    if(argc != 2){
        fprintf(stderr, "USAGE: comp source-file-name\n");
        exit(1);
    }
    if((infile = fopen(argv[1],"r")) == NULL) {
        fprintf(stderr,"Source file %s can't open.\n",argv[1]);
        exit(1);
    }
    if((outfile = fopen("a.asm","w")) == NULL) {
        fprintf(stderr,"Object file creation failed.\n");
        exit(1);
    }
    compiler();
    fclose(infile);
    fclose(outfile);
    exit(0);
}
```

- コンパイラ本体用ファイル

```
#include <stdio.h>
#include <getsym.h>
extern  TOKEN   tok;            /* The current token */
extern  FILE    *infile;        /* Source File */
extern  FILE    *outfile;       /* Object File */

void error(char *s);
void statement(void);

void compiler(void){
  init_getsym();
  getsym();
  if(tok.attr == RWORD && tok.value == PROGRAM) {
    getsym();
    if(tok.attr == IDENTIFIER) {
      getsym();
      if(tok.attr == SYMBOL && tok.value == SEMICOLON) {
        getsym();
        statement();
        if(tok.attr == SYMBOL && tok.value == PERIOD) {
          fprintf(stderr, "Parsing Completed.  No errors found.\n");
        } else error("At the end, a period is required.");
      } else error("After program name, a semicolon is needed.");
    } else error("Program identifier is needed.");
  } else error("At the first, program declaration is required.");
}

void error(char *s){
  fprintf(stderr, "%s\n", s);
  exit(1);
}

void statement(void){
  /* 式を読み込み，その式を計算し結果を表示するコードを生成する処理 */
}
```