

# 情報科学実験Ⅱ

## 2018 年度 修正履歴

日付	頁	実験番号	説明
----	---	------	----

copyright © 2013-18 静岡大学情報学部情報科学科	第 1 版	2014 年 7 月 31 日
	第 2 版	2015 年 9 月 24 日
	第 3 版	2016 年 3 月 25 日
	第 4 版	2017 年 3 月 22 日
	第 5 版	2018 年 9 月 12 日

担当者：

太田 剛 (ohta@inf.shizuoka.ac.jp)

増澤智昭 (masuzawa.tomoaki@shizuoka.ac.jp)

## 概 要

「情報科学実験Ⅱ」では、コンパイラを作成する。

本テキストで説明するコンパイラは、C言語に似た言語を入力言語とし、SEP-3 アセンブリ言語を出力する。このコンパイラの出力を、「機械語と計算機械」で作成したアセンブラに入力することで、SEP CPU 用の機械語が生成できる。こうして生成した機械語が、SEP ボード上で、もしくは SEP シミュレータ上で動作するようにするのが目標となる。

その際、入力言語の仕様を最初に完全に定義し、それに基づいて順々にコンパイラを作っていく方法が王道ではある。しかしながら、その方法は目に見える結果（これで確かにできているという実感）がなかなか出ないので、途中でいやになったり飽きてしまったりという弊害もある。

そこで、このテキストでは、まずはごく小さな仕様の言語のコンパイラを皆さんに与え、そのプログラムを理解してもらったところから始める。その後、少しずつ言語仕様を拡張してプログラミングを行い、自分の作ったコンパイラが生成した機械語を読み、実際に動かしてみる。機能追加が確実に行われたかどうかは、教員または TA が逐一確認しながら進めてゆく。そして、最終的には C 言語のサブセットにまで拡張していくという手順を踏む。このとき、「機械語と計算機械」で作成したアセンブラと同じ設計思想を用いることにしよう<sup>a</sup>。

さて、コンパイラの作成は、相当な量のプログラミングを必要とする。しかしながら、真面目に取り組めば 1 章（1 実験課題）を 3 コマで終われる程度の分量に分割してあるので、毎週着実に実施していけば問題なく完了できるはずである<sup>り</sup>。ただし、量が量なので、学期終了間際になってあわてて全部やろうとしても、他科目の試験やレポートのため、とてもやりおせるものではない。

着実に進んでいるかどうかを確認するため、毎回、担当教員、TA が課題チェックを行う。“第 N 週の課題チェックを第 N+1 週に受けて合格する” — ことが標準となるように、課題は全部で 14 ある。こちらの実験をさっさと終えて、年明けには「情報科学実験Ⅲ」に専念したい学生は、どんどん先に進んでかまわない<sup>ろ</sup>。誰がどこまで合格しているかチェックリストを科 2 実験室に貼り出すので、担当教員、TA に質問しにくい場合は、誰に尋ねればよいかわかるはずである。

なお、第 I 部は全員クリアすることが必須である。すなわち、第 I 部がクリアできないと「情報科学実験Ⅱ」の単位は出ない。第 I 部の実験課題は全部で 8 課題あるので、最悪でも 1 課題を 2 週以内にこなさないと間に合わない。授業最終日以降には、いっさい課題チェックを行わないので、授業期間中に合格レベルにまで達していないと、そのまま「不可」となる。肝に銘じること。第 II 部の各実験はオプションであり、そのクリア度合いによって成績が決まる。

<sup>a</sup>コンパイラもアセンブラと同様に「言語処理系」と呼ばれるカテゴリに属する。そのため、アセンブラの設計思想は、そのままコンパイラ作成でも利用することができる。

<sup>り</sup>皆さんに与えるプログラムは、23 クラス 700 行弱。本テキストの実験 14 まで太田が実施してみた結果は、全部で 100 クラス 4200 行ほどであった。単純計算して、1 課題あたり 250 行程度のプログラミングが必要となる。数字だけを見ると驚くかもしれないが、太田が 1 コマ 90 分以内（「3 コマ 5 時間」じゃないですよ!念のため）に実施できたという実績に基づいて課題を分割した。

<sup>ろ</sup>年内に全課題を終える学生は、毎年いる。

## 凡 例

このタイプの囲みには、文法が拡張 BNF で記述されている。例えば、

```
term ::= factor { (PLUS | MINUS) factor }
```

小文字の名前は非終端記号を、大文字の名前は終端記号を表す（以下、本テキストでは全てこのルールに従う）。また、大文字の名前は、字句解析部から返却されるトークン情報と一致する。この文法では、PLUS と MINUS は、それぞれ（後に示す）CToken.TK\_PLUS と CToken.TK\_MINUS を表すこととする。

クラス名.java

このタイプの囲みには、基本的にはプログラムが記述されている。例えば、

```
public class Register {  
    private int value;           // 現在このレジスタが記憶している値  
    private int preValue;       // ゲートを越えて到着したデータ  
  
    // 各種 getter, setter  
    public int    getValue()     { return value; }  
    protected int getPreValue()  { return preValue; }  
  
    Register() { value = 0; preValue = 0; }  
}
```

1. そしてこのタイプの囲みには、実験課題が記述されている。

# 目次

<b>第 I 部 必須実験</b>	<b>1</b>
<b>実験 0 : 加算式の計算コンパイラ miniC ver.00</b>	<b>3</b>
0.1 構文定義 . . . . .	3
0.1.1 LL(1) . . . . .	3
0.1.2 パッケージ lang . . . . .	4
0.2 字句解析部 . . . . .	5
0.2.1 字句の定義 . . . . .	5
0.2.2 字句解析用の状態遷移図 . . . . .	5
0.2.3 状態遷移図のプログラミング . . . . .	6
0.2.4 パッケージ lang.c . . . . .	6
0.3 構文解析部 . . . . .	7
0.3.1 パッケージ lang.c . . . . .	7
0.3.2 パッケージ lang.c.parse . . . . .	8
0.3.3 構文解析のプログラミング . . . . .	8
0.4 意味解析部 . . . . .	10
0.5 コード生成部 . . . . .	10
0.6 実験のソースコード管理に関して . . . . .	13
<b>実験 1 : 減算の導入 miniC ver.01</b>	<b>15</b>
1.1 構文定義 . . . . .	15
1.2 字句解析部 . . . . .	15
1.3 構文解析部 . . . . .	16
1.4 意味解析部 . . . . .	16
1.5 コード生成部 . . . . .	16
<b>実験 2 : アドレス値の導入 miniC ver.02</b>	<b>17</b>
2.1 構文定義 . . . . .	17
2.2 字句解析部 . . . . .	17
2.3 構文解析部 . . . . .	17
2.4 意味解析部 . . . . .	18
2.5 コード生成部 . . . . .	18
<b>実験 3 : 四則演算と符号の導入 miniC ver.03</b>	<b>21</b>
3.1 構文定義 . . . . .	21
3.2 字句解析部 . . . . .	21
3.3 構文解析部 . . . . .	22
3.4 意味解析部 . . . . .	22
3.5 コード生成部 . . . . .	22

<b>実験 4 : 変数参照の導入 miniC ver.04</b>	<b>25</b>
4.1 構文定義 . . . . .	25
4.2 字句解析部 . . . . .	26
4.3 構文解析部 . . . . .	26
4.4 意味解析部 . . . . .	26
4.5 コード生成部 . . . . .	27
<b>実験 5 : 変数への代入の導入 miniC ver.05</b>	<b>29</b>
5.1 構文定義 . . . . .	29
5.2 字句解析部 . . . . .	29
5.3 構文解析部 . . . . .	30
5.4 意味解析部 . . . . .	30
5.5 コード生成部 . . . . .	30
<b>実験 6 : 変数宣言の導入 miniC ver.06</b>	<b>31</b>
6.1 構文定義 . . . . .	31
6.2 字句解析部 . . . . .	31
6.3 構文解析部 . . . . .	32
6.4 意味解析部 . . . . .	32
6.5 コード生成部 . . . . .	35
<b>実験 7 : 条件判定の導入 miniC ver.07</b>	<b>37</b>
7.1 構文定義 . . . . .	37
7.2 字句解析部 . . . . .	37
7.3 構文解析部 . . . . .	38
7.4 意味解析部 . . . . .	38
7.5 コード生成部 . . . . .	38
<b>実験 8 : if 文、while 文の導入 miniC ver.08</b>	<b>41</b>
8.1 構文定義 . . . . .	41
8.2 字句解析部 . . . . .	41
8.3 構文解析部 . . . . .	41
8.4 意味解析部 . . . . .	42
8.5 コード生成部 . . . . .	42
<b>第 II 部 オプション実験</b>	<b>45</b>
<b>実験 9 : 局所変数が使用可能な miniC</b>	<b>47</b>
9.1 構文定義 . . . . .	47
9.2 字句解析部 . . . . .	47
9.3 構文解析部 . . . . .	47
9.4 意味解析部 . . . . .	47
9.5 コード生成部 . . . . .	48
<b>実験 10 : 引数なし関数が使用可能な miniC</b>	<b>51</b>
10.1 構文定義 . . . . .	51
10.2 字句解析部 . . . . .	51
10.3 構文解析部 . . . . .	52
10.4 意味解析部 . . . . .	52

10.5 コード生成部 . . . . .	53
<b>実験 11: 引数付き関数を使用可能な miniC</b>	<b>55</b>
11.1 構文定義 . . . . .	55
11.2 字句解析部 . . . . .	55
11.3 構文解析部 . . . . .	55
11.4 意味解析部 . . . . .	56
11.5 コード生成部 . . . . .	57
<b>実験 12: 複雑な条件判定が可能な miniC</b>	<b>59</b>
12.1 構文定義 . . . . .	59
12.2 字句解析部 . . . . .	59
12.3 構文解析部 . . . . .	59
12.4 意味解析部 . . . . .	60
12.5 コード生成部 . . . . .	60
<b>実験 13: エラー回復が可能な miniC</b>	<b>61</b>
13.1 エラーのレベル . . . . .	61
13.2 各エラーレベルの実装方法 . . . . .	61
13.2.1 致命的エラー . . . . .	62
13.2.2 文句を言うだけで事足りるエラー . . . . .	62
13.2.3 回復できるエラー . . . . .	62
<b>実験 14: 多少の最適化が可能な miniC</b>	<b>65</b>
14.1 レジスタ使用の最適化 . . . . .	65
14.2 定数の事前計算による最適化 . . . . .	66

第I部

必須実験



第 I 部では、コンパイラの作成技法の基本を一通り学習する。

皆さんには、「10 進数の足し算が記述できる言語」のコンパイラ（を記述したソースコード）を与える。これを、まず四則演算ができるように拡張する。その後、大域変数を導入して変数宣言と代入ができるようにする。このとき、単純な整数型のほか、ポインタや配列も導入する。最後に、条件判定、そして条件判定を使った各種の文が扱えるように拡張してゆく。

第 I 部が終わる頃には、およそ 60 クラス 3000 行程度のプログラムになる。

## テキスト全体の概要

最初に、このテキスト全体に書いてある内容をかいつまんで示しておこう。以下のことを頭に入れてから、テキストを読み進め実験を実施して欲しい。

コンパイラ作成のポイントは、(1) 入力ファイルから構文木を作ること、(2) 構文木を深さ優先で探索しながら、意味解析とコード生成に必要な仕事をこなしていくという点である。これは、すべての言語処理系<sup>1</sup>に共通する枠組みである。

- 字句解析部は、「単なる文字の列」である入力ファイルを読み、意味のあるまとまりごとに区切って「字句（トークン）の列」へと作り変える。
- 構文解析部は、字句列を読み、それらが与えられた構文定義（文法規則）にしたがって並んでいるかどうかを確認し、構文木を作る。
- 意味解析部は、構文木を深さ優先探索しながら、意味上の誤り（変数名の宣言がないとか、変数の使い方が違うとか...）がないかどうかをチェックする<sup>2</sup>。このとき、プログラム中に出てくる名前（識別子と呼ぶこともよくある）がどのような意味を持つのかを「記号表」で管理していく。
- コード生成部は、構文木を深さ優先探索しながら、行きがけ、通りがけ、帰りがけに、構文の意味するところを実施できるようなコードを作り出していく。

---

<sup>1</sup>「言語処理系」とは、単語の並び—この並びは、ある文法規則にしたがっている—を読み込んで、その意味するところを実施して結果を返すような処理をするプログラムである。だから、基本的にはすべてのソフトウェアは言語処理系であると言える。ただ一般には、結構複雑な文法規則を持つようなものに対して「言語処理系」という用語が使われる。インタプリタ・コンパイラは当然ながら、「機械語と計算機機械」でやったアセンブラもそうだし、Windows 系の「コマンドプロンプト」の仕事も、UNIX 系の「シェル」の仕事も言語処理系に分類される。自然言語処理（機械翻訳とか...）も、もちろんそういう構成になっている。

<sup>2</sup>現実には、構文解析部と同時並行で実施すること多い。皆さんに渡すプログラムでは別々に実施している。

# 実験0： 加算式の計算コンパイラ miniC ver.00

## 概要

皆さんに、10進数の加算だけを扱うコンパイラのソースコードを提供する<sup>0</sup>。

この ver.00 のプログラムには、プログラミング言語というほどの機能はない。しかし、コンパイラをどう作っていけばよいかを知るには、これで十分である。まずは、このソースコードをよく読んで、コンパイラをどのようにして作ればよいのかを知ることから始めよう。

## 0.1 構文定義

入力プログラムの文法が次のように定義されている、10進数を複数足すことが記述できる超簡易言語を考えよう<sup>1</sup>（EOF は、「機械語と計算機械」でも扱った、入力ファイルの終了を表す特殊な字句である）。

```
program      ::= expression EOF
expression   ::= term { expressionAdd }
expressionAdd ::= PLUS term
term         ::= factor
factor       ::= number
number       ::= NUM
```

まず手始めに、この言語のコンパイラ（と言うのもおこがましいが...）miniC ver.00 の解説から始めよう。

### 0.1.1 LL(1)

この実験では、プログラミングが比較的簡単にできることを重要視しているので、構文（文法）は LL(1) に従うものとする。ver.00 の構文もこれに従っている（各自、確認すること）し、以降の実験項目においても同様である。以降の実験において、構文を自由に決めてよい場合であっても、LL(1) からはずれないように注意すること。なお、LL(1) がどんなものであったかは「言語理論」の時間に学習済みなので、ここで繰り返すことはしない。必要だと思ったら、復習しておくように。

なお、教科書・参考書では first 集合とか director を使って厳密に議論してあると思うが（そして「言語理論」の授業でもそうであったはずだが）、LL(1) というクラスは、ようするに、規則に枝分かれ（選択肢）があるとき、そのどちらを選べばよいかを決定するのに、次の字句をひとつだけ読んでくれば完全に決められるような文法である。上の文法規則においては、選択肢が出現するのは `expression` の定義の 2 項目にある `{` のところが唯一である。つまり、繰り返すするかしないかを決定するには、次の字句をひとつ読み、それが `expressionAdd` の first に属する字句なのか、EOF なのかを判断すればよいということである。

<sup>0</sup>ただし、ソースコード自体には、ほとんど注釈（コメント）を付けていない。一方、この指導書には、コメントとして書き切れないほどの量の情報を載せてある。指導書を読んでからソースコードを確認すること。「コメントのないソースコードは、全然理解できません」などと授業アンケートに書かないように。

<sup>1</sup>term, factor, number が無駄な非終端記号に思えるかもしれないが、将来の拡張においてプログラムの修正を少なくするために、今からこうしておく。さらに `expression` に関して、`{}` を用いて 0 回以上の繰り返しで定義しているが、この定義だと、演算を右結合とするか左結合とするかは、構文解析時の構文木の作り方（プログラミングのやり方）によって決めることが可能となる。つまり、`1+2+3` を `((1+2)+3)` のように解析木を作ることも、`(1+(2+3))` のように作ることもできる。`expression ::= term PLUS expression` と定義すれば右結合に確定してしまうし、`expression ::= expression PLUS term` なら左結合に確定してしまう（ただし、後者は LL(1) ではないことに注意）。

### 0.1.2 パッケージ lang

コンパイラ的设计方針は、「機械語と計算機械」で実施したアセンブラのそれをほぼ踏襲するので、まずはその復習をしておこう。

「コンパイラ」も「アセンブラ」も、どちらも「言語処理系」と呼ばれるカテゴリに属する。つまり、何らかの（プログラミング）言語で記述されたものを、ファイルから読み出して処理するのである。このパッケージは、プログラミング言語の抽象的概念を集めたパッケージであり、このパッケージ内のクラス（主に抽象クラス）を継承して現実のコンパイラを実現する。

#### 入出力コンテキスト（クラス名：IOContext）

入出力ファイルの名前やストリームをまとめて管理するクラスである。入力ファイル名から、規則にしたがって出力ファイル名やエラー出力ファイル名を生成すること、のほか、以下に解説する多くのクラスで共通してこれらを利用しなければならない。

#### 字句タイプの定義（クラス名：Token）

「言語理論」の講義で学んだように、アセンブラやコンパイラ等の言語処理系は、入力ファイルから「文字」を読み出して組み立て、「字句」にして切り出す必要がある。このクラスは、字句へのアクセスルーチンを定義した抽象クラスである。この抽象クラスを継承して作る（具象）クラスにおいて、その他の字句タイプを定義して使うことになる。

#### シンプルな字句（クラス名：SimpleToken）

上記 Token クラスの具象クラス。切り出した「字句」がどのタイプなのか（識別子なのか、記号類なのか、...）を定義してある。ただし、このクラスで定義しているのは、識別子、数値、EOF（ファイルの終端）、不正な字句、の 4 種類だけである。コンパイラ用の字句クラス（後述）は、これを継承して作ってある。

#### 字句解析器（クラス名：Tokenizer）

字句切り出しルールにしたがって実際に字句を切り出してくるものを表す抽象クラス。既に読み出されてしまっている最新の字句を返す `getCurrentToken()` と、まだ読み込まれていない新しい字句をひとつ切り出す `getNextToken()` の 2 つのメソッドが重要。

#### 解析コンテキスト（クラス名：ParseContext）

入出力および構文解析ルールに関わるメソッド群と、エラーに関する扱い方が定義してある。現在の構文解析がどのようなコンテキスト（文脈）で行われているか（どのファイルを、どの字句解析器で解析中かなど）を管理している。

#### 文法規則定義（クラス名：ParseRule）

構文規則を解析するための抽象クラス。これを継承して作る具象クラスにおいて `parse()` メソッドを自分で記述し解析する。

#### LL(1) 文法規則定義（インタフェース名：LL1）

LL(1) 文法に従う構文規則を解析するためのインタフェース。これを継承して作る具象クラスにおいては、`isFirst()` メソッドを `static` に定義する必要がある。

### 抽象的コンパイラ定義（インタフェース名：Compiler）

これは、クラスではなくインタフェースである。コンパイラと呼ばれるものを作るときに、`ParseRule`（を継承したクラス）で必ず定義しなければならないメソッド群。ここには `semanticCheck()` と `codeGen()` が指定されており、それぞれ、意味解析用、コード生成用のメソッドである。

### 致命的エラー例外：（クラス名：FatalErrorException）

プログラム中で致命的なエラーが生じたときに投げる例外。main メソッドで catch するように作ること。

## 0.2 字句解析部

### 0.2.1 字句の定義

前述の構文定義に従えば、字句としては次のものが認識できるだけでよい。

- 数（NUM）： 本章では、10 進数が扱えれば十分。ただし、Java が扱う整数型の最大値・最小値を超えない範囲であること。
- 加算記号（PLUS）
- ファイル終了記号（EOF）

これらを認識するための字句解析プログラムをまず解説するが、「機械語と計算機械」で扱った `SimpleTokenizer` は単純すぎて、本実験が最終目標とする（ちょっと大きな言語仕様の）コンパイラ用としては使えない。そこで、少々複雑な字句でも完全に扱えるように、状態遷移（オートマトン）の概念に基づく字句解析用クラスを作成する。

### 0.2.2 字句解析用の状態遷移図

まずは、前述の字句 3 種類（数、加算記号、ファイル終了記号）を認識するための状態遷移図を書いてみよう。ここでは敢えて図を描かずに言葉だけで説明するので、「言語理論」の授業でやったことを思い出しながら各自で図を描きながら作業すること。

初期状態において、

- 空白文字（水平タブ、改行<sup>2</sup>、スペース）を読んだら、何も読まなかったのと同じ（初期状態を維持する）。
- 加算記号（+）を読んだら、加算記号を認識するための状態へ遷移して受理する。
- ファイル終了記号（EOF）を読んだら、ファイル終了記号を認識するための状態へ遷移をして受理する。この時の受理状態は、加算記号を受理する状態とは別の状態でなければならない。
- 数字<sup>3</sup>を読んだら、数を認識するための状態へ遷移する。その先では、数字を読んでいる限り同じ状態を維持する。数字以外のものが来たら、数を受理する。このとき、最後に読んだ文字は、ひょっとすると次の字句の先頭の文字かもしれないので、読まなかったことにしておく必要がある<sup>4</sup>。
- それ以外の文字を読んだら、不正状態へ遷移して受理する（不正な字句 `CToken.TK_ILL`）を受理する）。

<sup>2</sup>Windows では、`\r`（行頭へ戻す）と `\n`（行送り）の両方を処理することが必要。UNIX 系 OS では、`\n` だけで十分

<sup>3</sup>本テキストでは「数字」と「数」と「数値」を厳密に使い分けているので注意すること。「数字」は「数を表す 1 文字」のことで、「数」は「数字」の列で構成された並びのこと、「数値」は「数」が表している値のこと。

<sup>4</sup>例えば、入力文字列が `1234+56` であった場合、+を読んで初めて数（1234）が終わったことを認識できる。このとき+は既に読んでしまっているので、このまま何もせずに受理してしまうと、次に `getNextToken()` をしたとき、そこで最初に読み出される文字は 5 になってしまう。つまり、+が読み飛ばされてしまうことになる。それを避けるには、最後に読んだ数字以外の文字（この例では+）を読まなかったことにしてから受理しなければならない。そうすれば、次の `getNextToken()` では最初に+が読み出せる。「読まなかったことにする」のをどのようにして実現すればよいかは、アセンブラを作成するときに使った `SimpleTokenizer()` を見よ。今回も同様にプログラミングする。

1. 上記の状態遷移図を描いてみよ。

### 0.2.3 状態遷移図のプログラミング

状態遷移図をプログラミングするには、定石がある。次のプログラム片を読めば、その定石がわかるだろう<sup>5</sup>。

状態遷移図プログラミング定石.java

```
private CToken readToken() {
    char ch;          // 最後に読んだ文字
    int state;         // 状態番号
    boolean accept;    // 受理状態か?
    CToken tk;         // 受理した字句

    state = 0;         // 0: 初期状態
    accept = false;    // 受理状態じゃない
    while (!accept) { // 受理状態になるまで遷移を続ける
        switch (state) {
            case 0:     // 初期状態
                ch = readChar();
                if (ch == ' ') { state = 0; } // スペースを読んだら状態 0 へ
                else if (ch == '\t') { state = 0; } // 水平タブを読んだら状態 0 へ
                else if (ch == '\n') { state = 0; } // 改行を読んだら状態 0 へ
                else if (ch >= '0' && ch <= '9') { state = 1; } // 数字を読んだら状態 1 へ
                :
                break;
            case 1:     // 数を解析する状態
                ch = readChar();
                if (ch >= '0' && ch <= '9') { state = 1; } // 数字を読んだら状態 1 へ
                else { // 数字以外を読んだら受理
                    backChar(ch); // 最後に読んだ文字 (数字ではない文字) を読まなかったことにする
                    accept = true; // 受理状態にして
                    tk = new CToken(CToken.TK_NUM, ...); // 字句を返す
                }
                break;
            :
        }
    }
    return tk;
}
```

変数 `state` に状態番号を保持しておき、その状態番号のときに行うべき仕事を「`case 状態番号:`」のところに書く。それぞれの状態では、1 文字読んで場合分けしてそれぞれの遷移先を判断し、必要な作業を行ってから `state` に遷移先の状態番号をセットする。字句を受理したいときには、変数 `tk` をセットして、変数 `accept` を `true` にする。これでループから脱出して、`readToken()` を終了する。

### 0.2.4 パッケージ lang.c

ここで、与えたプログラムを少々解説しておく。このパッケージは、「C コンパイラ」と呼ばれるプログラムの主要要素を集めたパッケージである。

<sup>5</sup>このプログラムは手続き的である。しかも「超」が付くほどに。状態遷移を実装するオブジェクト指向的な方法はあるが、ここでは採用しない。「機械語と計算機械」で CPU の状態遷移を扱った部分を思い出すと想像できるだろう。

### 字句定義（クラス名：CToken）

字句として認識しなければならないカテゴリを定義する。SimpleToken クラスを継承して作り、ファイルの何行目、何文字目、どんな綴りか...といった情報も記録しておく。今回の ver.00 では、SimpleToken クラスに定義されていない TK\_PLUS（加算記号を表す字句）が定義される。これから先の実験で新しい字句を導入する場合は、すべてここに定義を書き込む。

### 字句の定義ルール（クラス名：CTokenRule）

切り出すべき文字列が、どの字句タイプになるのかを決めるルールを記述する。定義は次の通り。

```
CTokenRule.java  
  
public class CTokenRule extends HashMap<String, Object> { ... }
```

Object と指定しているところは、「機械語と計算機械」（アセンブラ）では TokenAssoc となっていた。今回は、ここに何を指定しても良いように Object としてある。が、典型的には、識別子と同じルール「英字で始まって英数字の繰り返し」で切り出すことのできるキーワード（int とか while とか）に対して、識別子であることとは違う情報を返すために使う。詳細は、第 6.2 節を参照のこと。

### 字句解析器（クラス名：CTokenizer）

C の字句解析器。先の「状態遷移図プログラミング定石」を用いて作ってある。

2. 与えられたソースコードの lang.c.CTokenizer を、自分の書いた状態遷移図と比べながら読んでみよ。
3. 理解できたら、lang.c.TestCToken を使って、どんな文字列がどの字句として認識されるかテストしてみよ。

## 0.3 構文解析部

構文解析部の仕事は、2 つある。ひとつは、字句解析部から字句を次々と受け取って、構文規則通りの字句列であることを確認すること。もうひとつは、構文規則にしたがって「解析木」を作って返すことである。

その方法を説明する前に、ここでも再び、設計方針をまとめておこう<sup>6</sup>。

### 0.3.1 パッケージ lang.c

#### C 構文解析コンテキスト（クラス名：CParseContext）

ParseContext を継承して作る。ver.00 ではたいしたことは何も記述していないが、将来ここにいろいろなことを記述する必要がある。

<sup>6</sup>ただし、ここでは「機械語と計算機械」のアセンブラでのそれとは、方針が若干異なるので注意すること。

## C 構文解析器 (クラス名: CParseRule)

ParseRule を継承 (extends) して作り、Compiler インタフェースと LL1 インタフェースを実装 (implements) する。このクラスでは、文法規則にしたがって構文を解析するためのプログラムを `parse()` に記述する。そしてこれを継承する各クラスは、構文解析の結果として作られる「構文木」の各節点を表すことになる。さらに、その節点における意味解析の作業を `semanticCheck()` に、コード生成の作業を `codeGen()` に記述する。

意味解析の際には、この節点が表しているのが、C のどの型の情報であるのかを推測し管理する必要がある。そのためのメソッド群がここに定義してある (後述)。

## コンパイラ (クラス名: MiniCompiler, TestCToken)

コンパイラの (いわゆる) メインメソッドと、字句切り出しがうまくできるかどうかのテスト用プログラム。

### 0.3.2 パッケージ lang.c.parse

(拡張) BNF 記述した構文ルールひとつひとつについて、ここに解析するためのクラスを置く。今回は、Program、Expression、Term、Factor、Number の 5 つのルールそれぞれを解析するように、5 つのクラスが置いてある。解析結果は、「構文木」の形で管理する (ようにプログラミングする)。構文木の構成法は、以下に解説する

### 0.3.3 構文解析のプログラミング

構文定義の BNF における非終端記号それぞれについてクラスを作り、プログラミングする (lang.c.parse 内)。例として Expression の主要部を解説する。

```

public class Expression extends CParseRule {
    // expression ::= term { expressionAdd }
    private CParseRule expression;

    public void parse(CParseContext pcx) throws FatalErrorException {
        // ここにやってくるときは、必ず isFirst() が満たされている
        CParseRule term = null, list = null;
        term = new Term(pcx);
        term.parse(pcx);
        CTokenizer ct = pcx.getTokenizer();
        CToken tk = ct.getCurrentToken(pcx);
        while (ExpressionAdd.isFirst(tk)) {
            list = new ExpressionAdd(pcx, term);
            list.parse(pcx);
            term = list;
            tk = ct.getCurrentToken(pcx);
        }
        expression = term;
    }
}

class ExpressionAdd extends CParseRule {
    // expressionAdd ::= '+' term
    private CToken plus;
    private CParseRule left, right;

    public ExpressionAdd(CParseContext pcx, CParseRule left) {
        this.left = left;          // 左部分木 (+の左に書いてあるもの) の保存
    }

    public void parse(CParseContext pcx) throws FatalErrorException {
        // ここにやってくるときは、必ず isFirst() が満たされている
        CTokenizer ct = pcx.getTokenizer();
        plus = ct.getCurrentToken(pcx);
        // +の次の字句を読む
        CToken tk = ct.getNextToken(pcx);
        if (Term.isFirst(tk)) {
            right = new Term(pcx);
            right.parse(pcx);
        } else {
            pcx.fatalError(tk.toExplainString() + "+の後ろは term です");
        }
    }
}

```

解析結果の構文木は、3 行目の変数 `expression` に保存する。これを `parse()` によってどうやって作るかという、次の通りである。まず、`parse()` にやってくる前の段階で、現在読み出されている字句が `expression` の `first` 集合に含まれることは確認されているので (Program クラス、MiniCompiler クラスの `parse()` を読んでみるとよい)、構文規則に従えばまず `term` が来るはずであるから、その解析をする。そのためには `new Term()` をして (つまり、`term` 用の節点を作って)、そいつに解析を任せる (`term.parse()` を呼ぶ)。戻ってくれば `term` の解析が終わって、その次の字句が現在読み込まれているはずであるから、それ (`getCurrentToken()` の結果) が `expressionAdd` の先頭記号であるかどうかを確認し (`ExpressionAdd.isFirst()`)、そうである間は、先ほどと同様に `new ExpressionAdd()` で `expressionAdd` 用の節点を作ってそいつに解析を任せる。任せるときには、解析済みの `term` の情報 ('+' 記号の左側の解析結果) を渡してやる<sup>7</sup>。解析から戻ってくれば、次の字句が読み出されているはずなのでそれを取り出し (`getCurrentToken()`)、繰り返しの先頭に戻る。繰り返しが抜けると (つまり、`expressionAdd` の先頭記号でない字句が読み込まれたら)、解析結果 (つまり、`expression` 以下の構文木) を変数 `expression` にしまっけて終了する。

<sup>7</sup>これを渡す理由は、加算を表す節点 (`ExpressionAdd`) が "+" 記号の左右両方の情報を管理するのが妥当であるから。そうしておけば、意味解析でもコード生成でも、仕事が楽にできる。`ExpressionAdd` が "+" 記号の左側の情報を持たずにおくと、その情報が欲しくなるたびに構文木の上位の節点に「お伺いを立てる」が必要になると比べて欲しい。



ExpressionAddで行っているのも構文規則に沿った解析作業である。注意すべき点は、コンストラクタに渡された left の値を解析結果として保存しておくこと (ExpressionAdd の 7 行目) と、term の先頭記号ではない字句が来たときにエラー処理 (fatalError()) をすることである。また、その節点が表す特徴的な字句 (この場合は '+') の出現場所を記録しておく、あとあと何かと都合が良いので、そうしてある (parse() 3 行目の plus 変数)。

## 0.4 意味解析部

意味解析の仕事は、semanticCheck() メソッドに記述する。そのためには、「型」と「型計算」(あるいは「型推測」) のことを理解しなければならない。

lang.c パッケージに CType クラスがある。このクラスは、miniC が扱える「型」に関するものである。ver.00 においては整数型 (int) しか必要ではないが、それだけだと何のことやらわからないので、ポインタ型 (int \*) まで含めて示しておいた。

この CType を使って「型計算」を行う。この型計算は、例えば、代入文において '=' 記号の左右両辺の型が一致しなければ代入できないとか、ポインタ同士を足すことはできないとかいったエラーを発見するために行う。

具体的なプログラムを ExpressionAdd クラスで見よう (ただし、ポイントになるところだけを抜粋し、エラー処理などの枝葉は削除してある)。

Expression.java

```
class ExpressionAdd extends CParseRule {
    private CParseRule left, right;
    :
    public void semanticCheck(CParseContext pcx) {
        // 足し算の型計算規則
        final int s[][] = {
            //      T_err      T_int
            { CType.T_err,   CType.T_err }, // T_err
            { CType.T_err,   CType.T_int }, // T_int
        };
        int lt = left.getCType().getType(); // +の左辺の型
        int rt = right.getCType().getType(); // +の右辺の型
        int nt = s[lt][rt]; // 規則による型計算
        this.setCType(CType.getCType(nt));
        this.setConstant(left.isConstant() && right.isConstant()); // +の左右両方が定数のときだけ定数
    }
}
```

ここでやっていることは、加算記号 ('+') の左にある式の型と右にある式の型から、結果の型を計算 (推測) することである。構文解析の結果は left と right 変数に格納したので、そこからそれぞれの型を取り出してきて (下から 5-6 行めの getCType()。これは CParseRule に定義されている)、その 2 つの情報から自分の節点の型を決める。その決め方は“規則表を引く”ことで行っている。あたりまえながら、int 型と int 型を足したら int 型になり、それ以外の組合せはすべてエラー型としている (配列 s[])。そしてその結果を、自分自身の型として記録する (下から 3 行めの setCType())。

また、左右の構文木がどちらも定数を表している場合に限って、自分自身も定数を表す情報を保持していることも記録している (下から 2 行めの setConstant())。これは、定数に代入していることを発見するとか、最適化のためにコンパイラ自身がある程度まで計算をしてしまう (例えば、a+3\*2 と書いてあったら a+6 とみなしてコンパイルしてしまう) とかいったことをするために行う。

## 0.5 コード生成部

さて、構文解析と意味解析が終われば、あとはコード生成の部分だけである<sup>8</sup>。それをどのように行っているか説明しよう。

<sup>8</sup>必須実験では最適化は扱わない。オプション実験には少しだけある。

コード生成に関するプログラムは、`codeGen` メソッドに記述する。言い換えれば、得られた解析木を深さ優先で走査しながら、行きがけ、通りがけ、帰りがけに必要な SEP-3 命令を出力してゆくように作る。このことは、`MiniCompiler` を見れば明らかである。

MiniCompiler.java

```
public class MiniCompiler {
    public static void main(String[] args) {
        String inFile = args[0]; // 適切なファイルを引数に絶対パスで与えること
        IOContext ioCtx = new IOContext(inFile, System.out, System.err);
        CTokenizer tknz = new CTokenizer(new CTokenRule());
        CParseContext pcx = new CParseContext(ioCtx, tknz);
        try {
            CTokenizer ct = pcx.getTokenizer();
            CToken tk = ct.getNextToken(pcx);
            if (Program.isFirst(tk)) {
                CParseRule parseTree = new Program(pcx);
                parseTree.parse(pcx); // 構文解析
                if (pcx.hasNoError()) parseTree.semanticCheck(pcx); // 意味解析
                if (pcx.hasNoError()) parseTree.codeGen(pcx); // コード生成
                pcx.errorReport();
            } else {
                pcx.fatalError(tk.toExplainString() + "プログラムの先頭にゴミがあります");
            }
        } catch (FatalErrorException e) {
            e.printStackTrace();
        }
    }
}
```

こうして呼び出された `Program` の `codeGen()` メソッドが何をしているかというと、次の通りとなっている。すなわち、構文解析のときにセットされた解析木（変数 `program`）に対して、`codeGen()` メソッドを呼び出す（つまり、式計算用コードを生成してくれと頼む）。ただし、その前に（行きがけに）コードを生成する開始番地を設定したり（. = 0x100）、式の計算用スタックを初期化したりし、後で（帰りがけに）CPU 停止命令を置いてアセンブル終了擬似命令を置く。

Program.java

```
public void codeGen(CParseContext pcx) {
    PrintStream o = pcx.getIOContext().getOutputStream();
    o.println(";;; program starts");
    o.println("\t. = 0x100");
    o.println("\tJMP\t__START\t; ProgramNode: 最初の実行文へ");
    // ここには将来、宣言に対するコード生成が必要
    if (program != null) {
        o.println("__START:");
        o.println("\tMOV\t#0x1000, R6\t; ProgramNode: 計算用スタック初期化");
        program.codeGen(pcx);
        o.println("\tMOV\t-(R6), R0\t; ProgramNode: 計算結果確認用");
    }
    o.println("\tHLT\t\t\t; ProgramNode:");
    o.println("\t.END\t\t\t; ProgramNode:");
    o.println(";;; program completes");
}
```

実際に計算をするコードを生成している `ExpressionAdd` の `codeGen()` を見てみよう。ここでは、計算用スタックを作業用に使って 2 数の加算ができるようなコード生成をしている。構文木の `left` と `right` にコード生成を依頼して、計算用スタックのトップに 2 つの値を残してもらい、帰りがけに、その 2 数を取り出して、足し、答えを積む<sup>9</sup>。

<sup>9</sup>「言語理論」で学習した、スタックを用いて式の評価を行うやり方を思い出して欲しい。「ポーランド記法」とか「逆ポーランド記法」とか「演算子順位文法」とかを学習したときにやったはずだ。

Expression.java

```

class ExpressionAdd extends CParseRule {
    private CToken plus;
    private CParseRule left, right;
    :
    public void codeGen(CParseContext pcx) {
        PrintStream o = pcx.getIOContext().getOutputStream();
        if (left != null && right != null) {
            left.codeGen(pcx); // 左部分木のコード生成を頼む
            right.codeGen(pcx); // 右部分木のコード生成を頼む
            o.println("\tMOV\t-(R6), R0\t; ExpressionAdd: 2数を取り出して、足し、積む<"
                    + plus.toExplainString() + ">");

            o.println("\tMOV\t-(R6), R1\t; ExpressionAdd:"
                    + plus.toExplainString() + ">");
            o.println("\tADD\tR1, R0\t; ExpressionAdd:");
            o.println("\tMOV\tR0, (R6)+\t; ExpressionAdd:");
        }
    }
}

```

最後に、この `left` と `right` に相当する `Number` クラスがやっていることを見てみよう。構文解析したときに字句として得た数（を表す字句）を `num` 変数に記録しておき、その数値をスタックに積む作業をしている。

Number.java

```

public class NumberNode extends CParseRule {
    private CToken num;
    :
    public void codeGen(CParseContext pcx) {
        PrintStream o = pcx.getIOContext().getOutputStream();
        o.println(";;; number starts");
        if (num != null) {
            o.println("\tMOV\t#" + num.getText() + ", (R6)+\t; Number: 数を積む<"
                    + num.toExplainString() + ">");
        }
        o.println(";;; number completes");
    }
}

```

以上で、10 進数の加算だけを扱うコンパイラ miniC ver.00 の解説は終わりである。

なお、ver.00 は足し算をする機能だけがあって、結果を出力する機能はまだない。SEP ボード上、あるいはシミュレータ上で動作を確認したい場合は、停止後の R0 の値を見れば確認できる。

4. 与えられたソースコードをよく読め。

5. テスト用入力ファイル（lang.c パッケージ内 test.c）の内容を変えてみて実行し<sup>10</sup>、どんなコードが生成されるか観察してみよ。

本章に関しては、コンパイラの設計とソースコードの構成を理解し読むこと、テスト用入力ファイルを書き換えてみてどんなコンパイル結果が得られるかを観察すること、そして、ふむふむ... と理解すること、がすべてである。初回実験の本当のメイン作業は、次章にある。

<sup>10</sup>Eclipse を使っている場合は、メニューバーの「実行 (Run)」から「実行構成 (Configuration...)」を選んで、「引数 (Argument)」タブの「プログラムの引数」を書く必要がある。

## 0.6 実験のソースコード管理に関して

これから最低でも 8 課題、全部で 14 課題に取り組むことになる。そこでは、どの時点のどのソースコードで何が可能でいて何が実現できていないかを、あなたがた自身が逐一管理していかなければならない。

本実験が扱うソースコード量程度であれば、自分の頭の中でそのすべてを管理できる範囲内にある（結構ギリギリだとは思うが）。だから、本節に書いてあることは自分には必要ないとする学生もいるだろうし、そう思う学生は（本テキストの課題説明のところに書いてある通り）、「今動いているプロジェクトごと、ソースコードを丸々コピーして保存しておく」手法で乗り切っても一向に構わない。だが、意欲のある学生は、こういったことをきっちり管理してくれる「バージョン管理システム」を使ってみるのもよいかもしれない。これを使えば、簡単な手続きで、自分が望む時点のソースコードを、いつでも取り出すことができる。バージョン管理システムを使うことの利点は、試しに行ってみた変更がうまくいかなかったとき、きちんと動いていた時点のソースコードに戻す作業が容易にできること、コピーを取っておかなくてよいので、Eclipse の Java プロジェクトが無用に増えるのを防ぐことができるという点である。この他、複数人でプロジェクトを運営する場合に最新のソースコードを共有するとか、バグの通知や修正を管理できるとか、まあ、いろいろと他の利点もある。

ただし、バージョン管理システムの設定をきちんと行うのは少々骨が折れるので、全学生必須ということにはしない。意欲および興味のある学生は、例えば次の URL などを参考にして使用してみることを勧める。Subversion も Git も、有名なバージョン管理システムの名前である。なお、サーバマシンもクライアントマシンも、どちらもあなた自身の PC にすればよい。

- バージョン管理システムとは
  - － 知らないで現場で困るバージョン管理システムの基礎知識  
<http://www.atmarkit.co.jp/ait/articles/1305/20/news015.html>
  - － サルでもわかる Git 入門～バージョン管理を使いこなそう～  
<http://www.backlog.jp/git-guide/> の「入門編」中「Git の基本」のところ
- Subversion を使う
  - － バージョン管理システムを使おう（Subversion 編）  
<https://www.agilegroup.co.jp/technote/subversion-first.html>
  - － Eclipse で Subversion を使う  
<https://www.ibm.com/developerworks/jp/opensource/library/os-ecl-subversion/>
- Git を使う
  - － Git 公式サイトのドキュメント  
<https://git-scm.com/book/ja/v1>
  - － サルでもわかる Git 入門～バージョン管理を使いこなそう～  
「バージョン管理システムとは」のところの URL を参照
  - － Eclipse で Git をクライアント用途として使用 (EGit)  
<http://www.kakiro-web.com/memo/eclipse-git-client.html>

上記全ての URL の存在は、平成 30 年 9 月 11 日に確認した。



# 実験 1 : 減算の導入 miniC ver.01

## 概 要

加算だけができるコンパイラ miniC ver.00 を、加減算を行えるコンパイラ ver.01 に拡張する。また、プログラム中にコメントを書いてもよいようにしよう。

### 1.1 構文定義

ver.00 の構文定義を少し拡張して、次のように変更しよう。どうということはない。足し算に加えて引き算ができるようにするだけである。ここに記載していない非終端記号には、何も変更がない。

```
expression    ::= term { expressionAdd | expressionSub }
expressionAdd ::= PLUS term
expressionSub  ::= MINUS term
```

### 1.2 字句解析部

今回は、引き算記号(‘-’)が認識できることのほか、コメントが書けるように字句解析部分を修正しよう。コメントは、次の 2 種類が扱えるようにしたい。

- /\* で始まって \*/ で終わるコメント。複数行にまたがるコメントが許される。
- // で始まって行末で終わるコメント。行末を越えて次の行にまでコメントが及ぶことはない。

1. 上記 2 種類のコメントに関して、状態遷移図を描いてみよ。「言語理論」の授業では、その状態遷移図が意外と難しいことを学習したはずだ。なお、コメントが終わる前に EOF が来たときどうすべきかについて、よく考えること。
2. 遷移図が描けたら、教員または TA に確認してもらうこと。
3. OK をもらえたら、Eclipse で miniCV00 プロジェクトのコピーを作って、miniCV01 プロジェクトと名前を付ける。本章のプログラミングは必ず miniCV01 プロジェクト上で行うこと。
4. 追加した状態遷移図 (2 種のコメントと ‘-’ 記号) をプログラミングせよ。
5. 教員から指示されたテストケースがうまく字句解析できるかテストせよ。字句解析部のテストには、TestCToken.java を使うと良い。

### 1.3 構文解析部

今回の修正は軽微で、`Expression` クラスを少し変えるだけで済む。だって、構文上はそこしか変更がないのだから。減算用の解析木節点として、`ExpressionSub` クラスを作ることになろう。`ExpressionAdd` のそれをコピーして少し修正するだけでよい。

6. 前述の構文定義に従って、`Expression.java` を修正せよ。

### 1.4 意味解析部

整数型しか使わないので、足し算も引き算もやることは同じである（ポインタを扱うことになると、足し算と引き算とでは違って来る）。今回はコメントだけを直しておけばそれで OK だ。

### 1.5 コード生成部

ここでは、`ExpressionSub` の `codegen()` メソッドを記述すればよい。`ExpressionAdd` のものを参考にすれば、難しくはないはずだ。

7. 足し算と引き算の両方をコンパイルできる miniC ver.01 を完成させよ。SUB 命令が、どちらからどちらを引く命令だったかよく考えて、正しいコードを生成するように！

8. 完成したら、教員または TA に確認してもらうこと。

## 実験 2： アドレス値の導入 miniC ver.02

### 概 要

数として、アドレス値が使えるようにする。単に“100”と書けば 10 進数・整数型の値を意味するが、“&100”と書けば（整数の 100 ではなく）“100 番地”を意味することにしよう。これに伴い、0xffe0 のような 16 進数表記、0472 のような 8 進数表記（いずれも C の作法）も導入する。

### 2.1 構文定義

ver.01 の構文定義を少し変更する。数の前に&を付けられるのが新しい機能である<sup>1</sup>。

```
factor    ::= factorAmp | number
factorAmp ::= AMP number
```

注：AMP='&'

### 2.2 字句解析部

1. '&' と 16 進数 8 進数が正しく解析できるような状態遷移図を描き、プログラミングしてみよ。Eclipse で MiniCV01 プロジェクトのコピーを作って、MiniCV02 プロジェクトと名前を付ける。本章のプログラミングは必ず MiniCV02 プロジェクト上で行うこと。
2. 16 ビットで表現できないほど大きな（小さな）数（10 進、16 進、8 進）が入力ファイルに書いてあった場合にはどう扱えばよいか？ 対処方法とアルゴリズムを考えてプログラムを修正せよ。
3. できた字句解析プログラムを TestCToken.java でテストせよ。

### 2.3 構文解析部

修正するところは、factor のところだけだ。&が来たら FactorAmp クラスを new して解析を依頼すればよい。

4. 前述の構文定義に従って、Factor.java を修正せよ。

<sup>1</sup> 「factor ::= [ AMP ] number」と定義するほうが楽なのに、なぜそうしないか？と考えるのはもっともだが、後々修正が加わることを考えると、この定義のほうがプログラミング上有利だからそうしてあるのだ。[ AMP ] number と定義すると、Factor 節点では“&”があるときとないときとで別の仕事（意味解析時もコード生成時も）をするようにプログラミングしなければならない。一方、factorAmpを導入すれば、ひとつの節点ではひとつのことだけに注目すればよい（FactorAmp では&についてだけ。Number では数のことだけ）。オブジェクト指向に限らずプログラミングするときは一般に、保守のことを考えて後者の戦略を取るのが常識である。



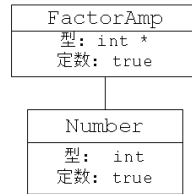


図 2.1: &amp;があった場合の factor の構文木

## 2.4 意味解析部

&の導入によって、新たにポインタ型が加わった。今回の実験は、これに関する意味解析の方法を考えることが肝である。

まずはポインタ型を CType クラスに加える必要がある (... が、それは与えたプログラムでは既にしてある)。その上で、加減算の型計算規則がどうなるかを考えて、`semanticCheck()` を修正しなければならない。

まずは、FactorAmp クラスの意味解析について考えよう。

FactorAmp においては、その（構文木における）下の節点が整数を表しているとき、その整数をポインタとみなすという操作をする（図 2.1 の「型」の部分参照）。したがって、下の節点が整数型 `int`<sup>2</sup> のときに、この節点の型を整数へのポインタ型 `int *`へと変える型計算規則を持つことになる。また、このようにして作り出したポインタは、必ず定数を表すことになるのも理解できるだろう<sup>3</sup>。

次は、加減算における型計算の扱いである。C では、ポインタ ± ポインタ、ポインタ ± 整数、整数 ± ポインタ<sup>4</sup>、整数 ± 整数が、それぞれどう扱われ、結果がどの型になると定めているかを調べ、その通りにプログラミングしよう。

5. FactorAmp、ExpressionAdd、ExpressionSub について、意味解析部をプログラミングせよ。

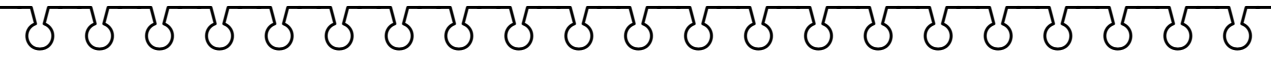
## 2.5 コード生成部

今回導入した演算 '&' については、どんなコードを生成すればいいだろう？ よくよく考えてみて欲しいのだが、実は、何も生成する必要がない。なぜなら、この演算子は、型を変えるだけの意味しか持たず、値を変えないからだ。... というわけで、今回はコード生成部でやるべき仕事はない。ただし、FactorAmp 節点の下にある節点のコードは生成してもらわなければならないので、子節点の `codeGen()` メソッドを呼ぶことを忘れてはならない。

<sup>2</sup>将来のバージョンでは配列である場合が出てくるが、今は考えない。

<sup>3</sup>同様に、将来のバージョンでは変数名や配列名に&を付けることができるようにするが、その場合もまた定数である。なぜなら、それらの変数が割り当てられるはずの番地は定まっており、プログラマが書き換えてよいものではないからだ。

<sup>4</sup>減算は可換（右と左を交換しても結果が同じ）な演算ではないので、左右を入れ替えた組合せも考える必要がある。

- 
6. 本章の最初に示した構文をコンパイルできる MiniC ver.02 を完成させよ。
  7. 教員から指示されたテストケースについてテストを行え。これは必要最低限のテストケースである。
  8. よく考えて他のテストケースを作成し、作ったテストケースに基づいてテストを行え。
  9. テストを終えたら、教員または TA に確認してもらうこと。



## 実験3： 四則演算と符号の導入 miniC ver.03

### 概 要

加減算に加えて乗除算ができ、かっこを用いて演算の優先順位を変えることができるようにしよう。

### 3.1 構文定義

ver.02 の構文定義をさらに少し変更する。(1) 符号 (±) を付けることができ、(2) 乗除算が許され、(3) かっこによる優先順位の変更ができる、... のが新しい機能である。構文定義がだんだん複雑になってきているが、LL(1) であることは各自確認しておくこと。BNF の上では、前章の `factor` を `unsignedFactor` に格下げ (?) して、符号が書けるようにしてある。

```
term          ::= factor { termMult | termDiv }
termMult      ::= MULT factor
termDiv       ::= DIV factor
factor        ::= plusFactor | minusFactor | unsignedFactor
plusFactor    ::= PLUS unsignedFactor
minusFactor   ::= MINUS unsignedFactor
unsignedFactor ::= factorAmp | number | LPAR expression RPAR
```

(注) LPAR='(', RPAR=')'

### 3.2 字句解析部

miniC ver.03 で新しく追加された字句は、`'*'`、`'/'`、`'('`、`')'` である。このうち 3 つは別段問題はないのだが、`'/'` だけはちょっと問題がある。コメントの開始記号 (の最初の文字) と同じだからだ。

1. `'/'` とコメントが正しく分離できるような状態遷移図を書いてみよ。
2. Eclipse で MiniCV02 プロジェクトのコピーを作って、MiniCV03 プロジェクトと名前を付ける。本章のプログラミングは必ず MiniCV03 プロジェクト上で行うこと。
3. 状態遷移図をプログラミングして、教員から指示されたテストケースでテストせよ。`'/'` とコメントが正しく分離できるか？

### 3.3 構文解析部

今回の修正は少し大掛かりになる。が、考え方は前の章と同じだ。`Factor.java`をコピーして`UnsignedFactor.java`を作り、両方を修正しよう。もちろん、`Term.java`も書き換える必要がある。

4. 前述の構文定義に従って、構文解析部を作成せよ。

### 3.4 意味解析部

考えるべきことは前章と同じである。ポインタの符号を反転することに意味があるのか？ ポインタと整数を掛けるとは？ ... など、いろいろと考えて規則を決め、そして実装すること。

5. 考えた結果に基づいて意味解析部を作成せよ。教員から指定されたテストケースをテストすること。

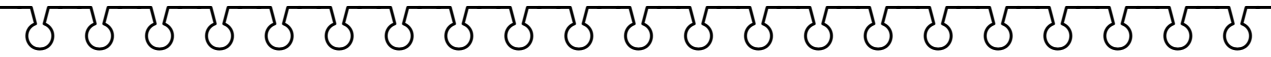
### 3.5 コード生成部

今回、解析木に追加した節点は、乗算、除算と符号（`'+'` と `'-'`）の節点である。このうち、符号の`'+'`は、生成すべきコードが何もない<sup>1</sup>。一方、`'-'`のときには符号反転のためのコードを生成する必要がある。スタックトップから値を取り出し、その値を 0 から引いた結果をスタックに積むコード、あるいは、スタックトップから値を取り出し、ビット反転して 1 を足した結果をスタックに積むコードを生成すればよい。いずれの場合もオーバーフローする恐れがある。どんな場合にそうなるのかを気にすることは大事なことはあるが、この実験ではそこまで考えてコードを生成する必要はない。

最後に、掛け算と割り算が残った（SEP には乗除算命令はないのだった）。実は、掛け算と割り算のサブルーチンは、「情報科学実験 III」で作成する予定である（掛け算については 2 年次の「コンピュータ設計 I」にも出てきた）。そこで、さしあたりは `JSR MUL` と `JSR DIV` 命令を生成しておくことにする<sup>2</sup>。

<sup>1</sup>それでも、子節点に向けてコード生成を依頼することはしなければならない。

<sup>2</sup>サブルーチン `MUL` と `DIV` は、引数をスタックから受け取り、結果をレジスタに返すものとする。ここまでに実装した演算子は、オペランドをスタックから受け取って結果をスタックトップに残しているの、サブルーチン `MUL` と `DIV` も、これと同じ作法に従う必要がある。すなわち、サブルーチンから戻ってきた後で、スタックに積んである引数を降ろす作業と、結果をスタックトップに残す作業とを行う命令を、ここで別途生成しておく必要がある。

- 
6. 本章の最初に示した構文をコンパイルできる MiniC ver.03 を完成させよ。
  7. 教員から指定されたテストケースについてテストすること。これらは必要最低限のテストである。これ以外にも必要だと思うテストを実行せよ。
  8. テストを終えたら、教員または TA に確認してもらうこと。
  9. (オプション) ここまでで、二項演算子 (加減乗除)、単項演算子 (符号、&) が出てきた。これら演算子に対して、`parse()`、`semanticCheck()`、`codeGen()` における共通作業はないだろうか。もしあれば、新しくクラスを作って共通部分をくくり出してまとめてみよう。こうすることで、プログラムの保守性が良くなる<sup>a</sup>。保守性が良くなれば、今後の修正もさらに楽にできるようになる。

---

<sup>a</sup>このような作業をリファクタリングと呼ぶ。



## 実験4： 変数参照の導入 miniC ver.04

### 概 要

(本実験項目は少々作業量が多く、時間がかかる。)

ここまでは数式が扱えるだけであったが、これからは数式の中に変数が使えるように miniC を改造する。まずは、単純な（扱いが容易な）大域変数が式の中に書けるようにしよう。

### 4.1 構文定義

いよいよ、変数を扱えるように改造を始めよう。そのためには、**変数を宣言**すること、**変数を参照**すること、**変数に代入**すること、の3つを扱えるようにしなければならない。このうち、代入は次の章で、宣言はさらにその次の章で行うとして、まずは式に変数を書いて参照できるようにしよう。ここに示していない非終端記号の定義は、前の章と変わらない。ここでも、構文が LL(1) に従うことは各自確認しておくこと。

```
unsignedFactor ::= factorAmp | number | LPAR expression RPAR | primary
factorAmp      ::= AMP ( number | primary )
primary        ::= primaryMult | variable
primaryMult    ::= MULT variable
variable       ::= ident [ array ]
array          ::= LBRA expression RBRA          (注) LBRA='[' , RBRA=']'
ident          ::= IDENT
```

識別子 (IDENT) 周りの記述ができるような構文要素 `primary` を追加した。識別子の前に `'*'` を付けたり、後に配列要素の参照が書けるようにしてある。この文法規則では、`'*' よりも [] のほうが優先`される（なぜそう断言できるのか?）。記述のしかたと演算子の結合の強さは C になっているが、プログラミングの無駄な複雑さを避けるため、`'*' はひとつだけ`、配列は 1 次元に限ることにする。

さて、今回の追加において、非常に微妙な点がかかわっている。以下の説明をよく理解して欲しい。

`factorAmp` において、「`'&'` 記号を `number` または `primary` の前に置ける」と定義されている。これは、「`&i`」とか、「`&a[4]`」のような記述ができることを意味するが、これはいったい何を意味しているのだろうか? C にならえば、これはそれぞれ、変数 `i` の割り当て番地、配列 `a` の 5 番目 (C においては 4 番目ではない。念のため) の要素が割り当てられた番地、をそれぞれ意味している。つまり、`primary`、`primaryMult`、`variable`、`ident` は、「値」を意味するものではなく、「番地」を意味するのである<sup>1</sup>。実験 2 において `&10` が 10 番地を意味するようにしたことと考え合わせると、`factorAmp` もまた番地を意味する。

一方で、`unsignedFactor` は、「値」を表している。その右辺に `number` や `expression` が出てきていること、つまり、`3+5` の意味するところを考えれば、これは明白だ。`3+5` が番地を表すなどとは考えられない。しかし `unsignedFactor` の右辺には、同時に `factorAmp` と `primary` も出てきている。つまり、ここ `unsignedFactor` の右辺において、意味の齟齬が生じている。これを解消しなければならない。

では、どうすればいいか?

`factorAmp` と `primary` の意味するものを、この時点で「番地」から「値」に変えるしかない。

<sup>1</sup> ようするに、「番地」情報を作り出すようなコードを生成しなければならない節点だということ。



primary において「番地」を「値」に変えるには、その「番地」から「値」を取り出せばよい。式の中に “i” とか “a[4]” と書いた場合、それは「番地」ではなく「値」を意味するのだから、これが当然の解答だ。したがって、「番地から値を取り出すためのためのコード」を生成をすることが必要となるが、それは unsignedFactor 節点で行うことはできない。そんなことをしたら、expression に対してもまた、同じコードが生成されてしまって困ったことになるからだ。そこで、「構文解析も意味解析もしないが、コード生成だけをする節点」を用意することで、これを解消する（下記 BNF の addressToValue がそれ）。

残る factorAmp が本当に微妙な要素である。結論を先に言ってしまうと、コンパイラの意味解析機能の立場からすると、これは「番地」から「値」への単なる型変換演算子にすぎない<sup>2</sup>。つまり、「&」の右側には番地情報が来るけど、これを番地情報としては使わずに値とみなすことにしますよ」ということである。よって、コード生成的には何もする必要がない（意味を変えるだけだから）。

以上のことから、BNF を次のように改訂するとプログラミングがうまくゆく。

```
unsignedFactor ::= factorAmp | number | LPAR expression RPAR | addressToValue
addressToValue ::= primary
```

## 4.2 字句解析部

新たに導入された字句は、'[', ']' と、識別子（英字で始まって英数字の繰り返しから構成される）である。これらを認識できるように字句解析部を変更せよ。識別子の終わり部分の状態遷移は、数の終わり部分の状態遷移と同じ考え方で作ればよい。

1. miniCV03 プロジェクトをコピーして miniCV04 プロジェクトを作成せよ。本章のプログラミングは、miniCV04 プロジェクトで行うこと。
2. 上記のことができるように字句解析部を改造し、教員の指示したテストケースをテストせよ。

## 4.3 構文解析部

3. 前述の構文定義に従う入力ファイルを解析できるように、プログラムを変更せよ。
4. 構文解析部分のテストケースを作成せよ。MiniCompiler.java の semanticCheck() と codeGen() の部分をコメントアウトして、作ったテストケースに基づいてテストを行え。ここで使用したテストケースを、この実験の最後に教員または TA にチェックしてもらう際に見せること。
5. うまくできたら、miniCV04 プロジェクトをコピーして、miniCV04a とでもしておくこと。こうしておけば、いつでもこの段階に戻ることができる。

## 4.4 意味解析部

配列の文法を導入したことにより、型の種類がこれまでの 2 倍に増えることになる。すなわち、int と int\* の他に int[]（整数の配列）と int\*[]（ポインタの配列）の 2 つの型が加わるようになった。これにより、加減乗

<sup>2</sup>プログラマの立場では、&はアドレス抽出演算子である。

除、&など、これまでに導入した演算子の意味解析部分はすべて見直す必要がある。また、配列を導入したことで、添字を記述する部分 — つまり、“[”と“]”の間に書く式 — の型が `int` でなければならないこと、配列名の識別子は `int[]` 型でなければならないこと<sup>3</sup>が要請される。さらに、ポインタ参照（例えば、`*p`）では、`p` が `int` であってはいけないことも要請されるようになる。

さて、注意深い学生は、この文法規則なら `&*var` と書けることに（既に）気付いていると思う。この意味するところは何か...を考えてみてほしい。すると、なぜこれがCでは許されていないのか、理解できるはずだ。ただし、文法規則でこれを禁止するのはかなり骨が折れる（大改訂が必要となる）。そこで、意味解析において、これを検出することにしよう。

Java には、`instanceof` という演算子が用意されていて、ある変数がどのクラスのオブジェクトを保持しているかを検査することができる。そこで、これを使うことにしよう。つまり、「`factorAmp` の子節点に `primary` がつながっているとき、その下には `primaryMult` クラスのオブジェクトが来てはいけない」ことを記述すればよい。

6. 前述の意味解析ができるようにプログラムを修正せよ。
7. 教員から指定された、意味解析部分のテストケースを実行せよ。これは必要最低限のテストケースである。構文解析部のテストの際につけた `MiniCompiler.java` の `semanticCheck()` の部分のコメントアウトをはずして、テストを行え。このとき、`Ident` クラスの `semanticCheck()` 部で、`setCType()` を用いて型を直書きする方法を採ること（変数宣言が扱えない現時点では、それ以外にテストのしようがない）。
8. うまくできたら、`miniCV04` プロジェクトをコピーして、`miniCV04b` とでもしておくこと。こうしておけば、いつでもこの段階に戻ってくることができる。

## 4.5 コード生成部

変数の参照をするには、変数が割り当てられているアドレス値が必要となる。今回は大域変数を扱うことにしたので、アセンブラの表記法によれば、変数の識別子が `VAR` であれば `#VAR` でアドレス値が取り出せる<sup>4</sup>。したがって、変数を参照するには、`#VAR` を使ってこのアドレスから値を取り出してスタックに積むコードを生成するようにすればよい。そのためには、この節点には識別子の綴りを保存しておく必要がある。

Ident.java

```
CToken ident;
public void parse(CParseContext pcx) throws FatalErrorException {
    CTokenizer ct = pcx.getTokenizer();
    CToken tk = ct.getCurrentToken(pcx);
    ident = tk;                                // ここで綴りを保存しておく
    tk = ct.getNextToken(pcx);
}
public void codeGen(CParseContext pcx) {
    PrintStream o = pcx.getIOContext().getOutputStream();
    o.println(";;; ident starts");
    if (ident != null) {
        o.println("\tMOV\t#" + ident.getText() + ", (R6)+\t; Ident: 変数アドレスを積む<"
                + ident.toExplainString() + ">");
    }
    o.println(";;; ident completes");
}
```

<sup>3</sup>C では `int*` であってもよいことにしているが、この実験では許さないことにする。許してしまうと、あとの実験で大変な思いをすることになる。

<sup>4</sup>当然ながら、どこかで `VAR: .WORD 0` のような擬似命令が必要となるが、それは変数宣言の話なので詳細は次々章に譲る。

識別子の前に '\*' が付いていたら、その変数はポインタと解釈する。ということは、単なる変数の扱いだけではなく、一度間接参照して値を取り出す必要があるということだ。

Primary.java

```
class PrimaryMult extends CParseRule {
    // primaryMult ::= '*' variable
    private CToken op;
    private CParseRule child;

    public void codeGen(CParseContext pcx) throws FatalErrorException {
        PrintStream o = pcx.getIOContext().getOutputStream();
        if (child != null) {
            child.codeGen(pcx);
            o.println("\tMOV\t-(R6), R0\t; PrimaryMult: アドレスを取り出して、内容を参照して、積む<"
                    + op.toExplainString() + ">");
            o.println("\tMOV\t(R0), (R6)+\t; PrimaryMult:");
        }
    }
}
```

それとは反対に、識別子の前に '&' が付いているときは、#VAR の値（アドレス値）そのものが必要なので間接参照してはいけない。

配列の扱いについては、ここでは述べない。どうすればよいのか、各自で考えよ。

9. 以上のことが実現できる MiniC ver.04 を完成させよ。

10. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行い、コード生成が正しいか確認せよ。

11. テストを終えたら、教員または TA に確認してもらうこと。終わったら、miniCV04a や miniCV04b プロジェクトは削除してよい。

## ボード上、あるいはシミュレータ上で動かしてみたい学生に対する補足

今のままでは、アセンブルができない。なぜなら、変数用の領域を確保していないからだ。そこで、Program クラスの codeGen() メソッド内で、JMP \_START の直後に、変数用の領域を確保するための “.word” や “.blkw” を出力するように書き換えておくと良い。これでアセンブルできるコードになる。

## 実験5： 変数への代入の導入 miniC ver.05

### 概 要

ここでは、変数へ代入できるように「代入文」を実現する。

### 5.1 構文定義

まずは、代入文を使える構文を定義しよう。ここに示していない非終端記号の定義は、前の章と変わらない。ここでも、構文が LL(1) に従うことは各自確認しておくこと。

```
program      ::= { statement } EOF
statement    ::= statementAssign
statementAssign ::= primary ASSIGN expression SEMI      (注) ASSIGN='=', SEMI=';'
```

プログラムには、代入文 (assignment statement) がいくつも書ける。statement という一見無駄な非終端記号の定義があるのは、この後、文の種類が増えていくからだということは、理解できると思う。

手続き型プログラミング言語を学び始めて最初に違和感を覚えるのは、代入記号に (それまで何年もの期間、左右の「値」が等しいことを意味するのだと洗脳されてきた、数学記号の) '=' を用いることだ。そして、その次は (多くの初学者は気が付いてもないことが多いのだが)、“i = i” と書いたときに、左辺の “i” と右辺の “i” とでは意味が違うということだ。つまり、左辺に表れるときには入れ物に付けた「名札」としての “i”、言い換えると、入れ物を特定するためのものである。これに対して、右辺に表れるときには、それまでの常識通りのもの、言い換えると、入れ物の中にしまっている「値」を意味することになっている。プログラミング言語においては、左辺の “i” は決してその中にしまっている「値」を意味するのではない。

こんな話をここでするのは、ほかでもない。代入文の左辺に primary が指定してあることの妥当性を考えて欲しいからだ。前章の構文定義のところで詳しく解説した通り、primary は「番地」を表すものだったから、入れ物を特定する「名札」の機能をちゃんと持っている。すなわち、代入文の左辺にあって、なにも問題はない。情報科学科の学生ならば、左辺が expression と定義されていた場合の異常さが直感的にわかるようであって欲しい。

### 5.2 字句解析部

新しく導入された字句は、'='、';' の2つである。

1. miniCV04 プロジェクトをコピーして miniCV05 プロジェクトを作成せよ。本章のプログラミングは、miniCV05 プロジェクトで行うこと。
2. 上記のことができるように字句解析部を改造し、テストせよ。

### 5.3 構文解析部

前述の構文規則を解析するプログラムを作ろう。Program クラスの中でたくさんの文を管理しなければならないのが少々問題となるが、手っ取り早いのは `java.util.ArrayList` クラスを使うことだ。

3. 前述の構文定義に従う入力ファイルを解析できるように、プログラムを変更せよ。
4. よく考えてテストケースを作成し、テストケースに基づいてテストを行え。MiniCompiler.java の `semanticCheck()` と `codegen()` はコメントアウトしておくこと。構文解析部が正しく実行できることを確認せよ。うまくできたら、miniCV05 プロジェクトのコピーを作っておくこと。いつでも戻れるように。

### 5.4 意味解析部

今回の意味解析は、2つのことを確認するだけだ。ひとつは、代入記号（`=`）の左辺と右辺で型が一致していることを確認する。もうひとつは、定数には代入できないのを確認することだ。後者には、各節点において行ってきた `setConstant()` の結果を使えばよい。また、文の節点では、この節点の情報を上の節点に渡す必要がない（だって、使うところが上にはないのだから）。そのため、`setCType()` も `setConstant()` も、行う必要はない。

5. 上記のことを実現せよ。
6. 教員から指定されたテストケースをテストせよ。

### 5.5 コード生成部

まず、代入文の左辺に関して、どんなコードを生成しておけばよいか考えよう。`primary` は「アドレス」をスタックに積んでおいてくれた。その後、右辺の式（`expression`）は、スタックに式の計算値を残しておくようなコードを生成してくれるのだった。すると、代入文では、スタックトップから（右辺の）式の値を取り出し、さらに（左辺の）書き込むべきアドレスを取り出し、そこへ実際に書き込むコードを生成すればよいことになる。

7. 本章の最初に示した構文をコンパイルできる MiniC ver.05 を完成させよ。
8. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。
9. テストを終えたら、教員または TA に確認してもらうこと。

今回は作業量が少ないので、次章の構文解析部のあたりまで進んでおくことを勧める。次章はかなりの作業量がある。

## 実験6： 変数宣言の導入 miniC ver.06

### 概 要

ここでは、「変数は使う前に宣言しなければならない」ようなコンパイラを作る。これによって、ようやくプログラミング言語らしくなってくる。

### 6.1 構文定義

まずは、宣言を書ける構文を定義しよう。プログラムには変数宣言と代入文をそれぞれ0個以上書け、配列宣言では大きさを数で指定する必要がある。宣言では、変数の型を示すことと、定数を定義できるようにしよう。

```
program      ::= { declaration } { statement } EOF
declaration ::= intDecl | constDecl
intDecl      ::= INT declItem { COMMA declItem } SEMI
constDecl    ::= CONST INT constItem { COMMA constItem } SEMI
constItem    ::= [ MULT ] IDENT ASSIGN [ AMP ] NUM
declItem     ::= [ MULT ] IDENT [ LBRA NUM RBRA ]
```

文の構文定義とちがって、今回は1つの非終端記号に多くのことが詰め込まれている。それは、文の場合には演算子ごとに異なる仕事（意味解析とコード生成）が必要だったため、構文木の節点をそれぞれ別のものにしておくのが都合が良かったからである。そのため、ひとつの非終端記号では（可能な限り）ひとつの演算子だけを扱うようにしてあった。

それに対して、宣言部では異なる仕事をする必要がない（そもそも、実行コードを生成しない。擬似命令による変数領域確保はするかもしれない）ので、非終端記号を多くしないでおく。

### 6.2 字句解析部

新しく導入された字句'int'と'const'は、識別子と同じ状態遷移規則によって切り出せる。そこで、識別子を受理する状態において、切り出した文字列がintであった場合にはCToken.TK\_INT型の字句を返すようにプログラミングする必要がある。そのためには、CTokenRuleにおいて次のようにこれを登録し、

```
public class CTokenRule extends AbstractTokenRule {
    public CTokenRule() {
        put("int", new Integer(CToken.TK_INT));
        put("const", new Integer(CToken.TK_CONST));
    }
}
```

CTokenizerでは、識別子の受理なのか、それ以外のキーワードの受理なのかを決める、こんなロジックを組んでおけばよい。

CTokenizer.java

```

case 20:    // 識別子を切り出すための状態
:
// 識別子を切り出す仕事が終わったら
String s = text.toString();
Integer i = (Integer) rule.get(s);
// 切り出した字句が登録済みキーワードかどうかは i が null かどうかで判定する
tk = new CToken(((i == null) ? CToken.TK_IDENT : i.intValue()), lineNo, startCol, s);
accept = true;
break;

```

1. miniCV05 プロジェクトをコピーして miniCV06 プロジェクトを作成せよ。本章のプログラミングは、miniCV06 プロジェクトで行うこと。
2. 上記のことができるように字句解析部を改造せよ。

## 6.3 構文解析部

前述の構文規則を解析するプログラムを作ろう。

3. 前述の構文定義に従う入力ファイルを解析できるように、プログラムを変更せよ。
4. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。うまくできたところで、プロジェクトのコピーを作っておくこと。いつでもこの時点まで戻れるように。

## 6.4 意味解析部

識別子（変数や定数名、関数名など）が宣言されているときには、次のようなことを調べる必要がある。(1) 宣言が複数あるとき、同じ識別子で型違いの別の宣言がされていないか、(2) 文中で使われている識別子は、宣言されたものかどうか、(3) 宣言の方法と文の中での使い方とが一致しているかどうか（例えば、配列名に代入はできないし、整数型変数の前に '\*' は付けられない）、(4) 識別子に対して許されている演算かどうか（例えば、ポインタ変数を 2 倍することに意味を見出すことは難しい）。

これを実現するためには、宣言された識別子の綴りと型を記録しておき、式を解析するときにこれを参照してチェックすることが行われる。識別子と型を記録しておくものは「記号表」、チェックのことは「意味解析」と一般に呼ばれている。本章以前の意味解析部では、(3)<sup>1</sup> と (4) をチェックしてきた。本章では、「記号表」を導入し (1) と (2) を実現しよう。

<sup>1</sup>ただし、宣言の構文規則を導入していなかったため、4.4 節で指示したように、「Ident クラスの semanticCheck() 部で、識別子の型を直書きする方法」をもって、宣言したのと同じ効果を持たせるようにしていた。

## 記号表

記号表は、宣言された識別子に関する情報を登録しておくものである。一般に、その識別子が宣言されたときの型やコード生成時に必要な情報（割り当てられるアドレス等）を、識別子の綴りに結びつける。

「記号表」は綴り（常識的には `String` で実現する）を用いて登録と検索をする。そのための抽象クラス `lang.SymbolTable`（記号表）と `lang.SymbolTableEntry`（記号表への格納情報）を用意しているので、これを使う。`SymbolTable` は次のように定義されている。

SymbolTable.java

```
public abstract class SymbolTable<E extends SymbolTableEntry> extends HashMap<String, E> { ... }
```

この読み方は、「`String` と “`SymbolTableEntry` を継承する `E` クラス” との `HashMap` として `SymbolTable` が作られる」ということである。

これらを使って、`CSymbolTable` と `CSymbolTableEntry` を定義することにしよう。

CSymbolTableEntry.java

```
public class CSymbolTableEntry extends SymbolTableEntry {
    private CType    type;        // この識別子に対して宣言された型
    private int      size;        // メモリ上に確保すべきワード数
    private boolean  constp;      // 定数宣言か？
    private boolean  isGlobal;    // 大域変数か？
    private int      address;     // 割り当て番地

    public CSymbolTableEntry(CType type, int size, boolean constp, boolean isGlobal, int addr) {
        this.type = type;
        this.size = size;
        this.constp = constp;
        this.isGlobal = isGlobal;
        this.address = addr;
    }

    public String toExplainString() { // このエントリに関する情報を作り出す。記号表全体を出力するときに使う。
        return type.toString() + ", " + size + (constp ? "定数" : "変数");
    }
    :
}
```

`CSymbolTableEntry` クラスは、識別子に対して宣言された型の情報、メモリ上に確保すべきワード数（整数・ポインタなら 1、配列ならその要素数）、定数宣言なのか変数宣言なのかを区別する情報、大域変数か局所変数かを区別する情報、そして割り当て番地<sup>2</sup>を保持する。

必須実験では、変数はすべて大域変数の扱いだが、将来局所変数を導入したい（実験 9）。そこで、`CSymbolTable` クラスの `private` 変数として `global` と `local` の 2 つを用意する。もちろん、`global` は大域変数用であり、`local` は局所変数用である<sup>3</sup>。ただし、本章では `global` の方しか使わない。

<sup>2</sup>この情報は、局所変数を扱うようになるまで使うことがない。大域変数の割り当て番地は、（コンパイラが `.word` や `.blkw` を生成しておきさえすれば）アセンブラが計算してくれる。

<sup>3</sup>もともとの C がそうであるように、局所変数宣言を何重にもネストできるようにすることを考える学生がいるかもしれない。その場合は、複数の `HashMap` をスタック状に管理する必要がある。そのような工夫をしたい者は、自由に工夫してよい。ただし、局所変数のアドレス算出には特殊なアルゴリズムが必要になるので、少々難しい。



CSymbolTable.java

```

public class CSymbolTable {
    private class OneSymbolTable extends SymbolTable<CSymbolTableEntry> {
        @Override
        public CSymbolTableEntry register(String name, CSymbolTableEntry e) { return put(name, e); }
        @Override
        public CSymbolTableEntry search(String name) { return get(name); }
    }

    private OneSymbolTable global;    // 大域変数用
    private OneSymbolTable local;    // 局所変数用
    // private SymbolTable<CSymbolTableEntry> global;    // こう書いても、もちろん OK
    // private SymbolTable<CSymbolTableEntry> local;    // (同上)
    :
}

```

最後に、`const` つまり定数の扱いについて補足しておく。定数は、変数と同じ扱いをするとプログラミングが楽である。つまり、変数と同様に領域を確保し、初期値として定数値を入れておく。ただし、そこに書き込むことはできないという点だけが変数と異なる。意味解析時にそのチェックが簡単にできるように、上記の `constp` が用意してある。

また、配列名は定数であることに注意せよ。配列名には値を代入できないことは、これで簡単にチェックできる。

## 意味解析の方法

これらのデータ構造を使って次のことを行う。

- 構文解析時に、識別子の宣言に対してこれらの情報を記号表に登録する。識別子の二重定義のチェックもここでやる（検索してみてヒットすれば、二重定義だとすぐわかる）。
- 文の中で変数を使用する時に（これはつまり、`Ident` クラスにおいて... という意味）、記号表に登録されている情報を `setCType()` と `setConstant()` を用いて解析木に引き渡して、解析木を上にかのぼりながら意味解析を行う。

これからわかるように、記号表に情報を登録するのは構文解析時であり、それを使うのは意味解析時である。

実はここに大きな問題がある。何が問題なのかわかるだろうか？

問題は、構文解析が終了すると記号表は消えてなくなることだ。特に、局所変数については、有効範囲（スコープ）をはずれると、記号表ごと消さなければならない。残しておいたのでは、見えないはずの識別子が見えてしまうことになる。つまり、構文解析時でさえ、消えてなくなる記号表（および、そこに登録されている変数情報）があるのである。上記の第 2 項「記号表に登録されている情報を」と書いてあるところでは、実際に記号表を検索しても必要な情報にはヒットしない（ことが多々ある）のだ。

それを回避する方法はひとつしかない。`Ident` クラスの構文解析時には、宣言された識別子かどうかをチェックしなければならない。そのとき必然的に記号表を検索する。見つかった場合には `CSymbolTableEntry` クラスの情報が戻ってくるので、これを `Ident` クラスの `private` 変数として記録しておくことだ。意味解析のときには、記号表を検索するのではなく、`private` 変数を参照して仕事をする。これしか回避の方法がない。

本節「意味解析部」で解説したことは、実は `parse()` メソッドで行う（意味解析のための）準備作業のほうが多い。特に、`Declaration` 以下のクラスでは、`semanticCheck()` においてやることは何もない。

5. 記号表を実現する2つのクラスを定義し、記号表への識別子登録・検索のメソッドを作成せよ。同一名の重複登録ができないようにするのを忘れないこと。また、記号表全体を表示するメソッドが `lang.SymbolTable` にあるので、デバッグ等に使うと良い。
6. これを使って、構文解析時に識別子の重複宣言や未定義使用をチェックするように変更せよ。記号表は、`CParseContext` クラスに保持させるのがよい。
7. 宣言した情報にしたがって意味解析できるように、プログラムを変更せよ<sup>a</sup>。
8. 宣言通りの意味解析ができるかどうか、教員から指示されたテストケースでテストを行え。

<sup>a</sup>定数宣言においては、宣言できるのは整数またはポインタ1ワードなので、`SymbolTableEntry` の `size` は1に決まっている。そこで、(本当は良いことではないのだが) この場所を流用して、定数値をここに置くようにしておくのが楽だ。

## 6.5 コード生成部

ここでは、変数宣言に対してどのようなコードを生成すればいいのかを考える<sup>4</sup>。ここまですっと、変数は大域変数の扱いだった。すると、変数が宣言されたときは、メモリ上にその変数用の領域を確保すればよい。そして、その領域には識別子と同じ綴りのラベルを付けておくことで、参照を容易にしておく。領域確保にはSEP-3の擬似命令 `.WORD` と `.BLKW` が使える。単変数、ポインタ変数の場合は「ラベル: `.WORD 0`」を生成し、1ワードを確保して初期値0を設定すればよく、配列、ポインタ配列の場合は「ラベル: `.BLKW 数`」を生成し、「数」ワード分の領域を確保すればよいということだ。

9. 本章の最初に示した構文をコンパイルできる MiniC ver.06 を完成させよ。
10. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。
11. テストを終えたら、教員またはTAに確認してもらうこと。うまくできたら、本章の途中でコピーしておいたプロジェクトはすべて削除してよい。

<sup>4</sup>代入文のコード生成は、既に終わっている。



## 実験 7： 条件判定の導入 miniC ver.07

### 概 要

代入文に加えて、条件分岐文、繰り返し文を導入しよう。その前提として、条件判定を書けるようにするのが、本章の目的である。

### 7.1 構文定義

条件判定に関しては、現実のプログラミング言語ではかなり複雑なことが書けるが、この実験ではそこまでやる必要はない。ごく基本的な判定ができるようにするだけでよい。もし頑張っているいろいろやってみたい学生がいれば、実験 12 のオプション実験で実施すること<sup>1</sup>。

```
condition      ::= expression ( conditionLT | conditionLE | conditionGT
                                | conditionGE | conditionEQ | conditionNE )
                                | TRUE | FALSE
conditionLT    ::= LT expression
conditionLE    ::= LE expression
conditionGT    ::= GT expression
conditionGE    ::= GE expression
conditionEQ    ::= EQ expression
conditionNE    ::= NE expression
```

(注) LT='<', LE='<=', GT='>', GE='>=', EQ='==', NE='!='

### 7.2 字句解析部

条件判定演算子はすべて新しい字句であるから、CTokenizer で受理できるように変更する。また、true と false はキーワードの扱い (int や const がそうであったように) とする。

1. miniCV06 プロジェクトをコピーして miniCV07 プロジェクトを作成せよ。本章のプログラミングは、miniCV07 プロジェクトで行うこと。
2. 上記のことができるように字句解析部を改造せよ。
3. 教員から指定されたテストケースを用いて、字句解析部をテストせよ。

<sup>1</sup>例えば、not, and, or や、'( ' と ') ' による優先順位の変更など。

## 7.3 構文解析部

今回は特に説明することはない。

## 7.4 意味解析部

さて、比較演算を実行した結果として何が返ってくるか（スタックトップに何を残すか）について、ここで決めなければならない。C では、比較演算も「式」の中に書けることになっているので、比較の結果として整数を返すと定められている<sup>2</sup>。しかしながら、その考え方はプログラミング言語としては一般的ではないので、ここではブール値が返ってくることに決めよう。別の言い方をすれば、`condition` は `expression` とは異なるということだ。そのため、`CType` に、`T_bool` という型をひとつ追加する。比較演算子用節点では、2 つの `expression` の型が一致していることを確認して、OK なら自節点をこの型にセットするように意味解析をすれば OK ということだ（後述する `ConditionLT` を参照）。

4. 前述の構文定義に従う入力ファイルを解析でき、意味解析ができるように、プログラムを変更せよ。
5. `condition` をテストできるような `main` プログラムを作れ。`MiniCompiler.java` をコピーして、`Program` クラスを `new` するのではなく、`Condition` クラスを `new` するように直せばできあがる。
6. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。

## 7.5 コード生成部

比較演算のときに、何をコード生成しておけばよいかを考えるのは少々難しい。

最終的には、先に示した通り、比較演算の実行結果として、スタックトップにはブール値（つまり `true` か `false` かを表す値）が残るようにする。つまり、スタックトップから 2 つの値（演算子の右と左にある `expression` が作り出す 2 つの値）を取り出し、比較して、`true` か `false` のどちらかをスタックトップに残しておく。

ここで、`true` と `false` を表す値を決めることにしよう。これはどのように決めてもいいのだが、それぞれ 1 と 0、あるいは 1 と -1 に決めるのが、いろいろな面を考慮して都合が良い。比較演算子に対応するコードを実行した後では、スタックトップの値を（`MOV` 命令を用いて）取り出して条件分岐をすることになるはずだ。SEP-3 の条件分岐命令は `BRN`、`BRZ`、`BRV`、`BRC` の 4 つあるが、`BRV`、`BRC` が参照する V ビット、C ビットは、`MOV` 命令ではセットされることはない。その一方で、`BRN`、`BRZ` が参照する N ビットと Z ビットは、`MOV` するだけで、その値に応じて適切にセットしてくれる。真を 1 で偽を -1 に決めておけば `BRN` 命令を使って偽のときに分岐できるし、1 と 0 に決めれば `BRZ` 命令を使って偽のときに分岐ができることになる。

最後に残ったのは、その 2 値のどちらをどうやってスタックに残すようにコード生成をするか... だが、これはサンプルコードを見てもらえばすぐに理解できるだろう。この例では、真を 1 で偽を 0 で表すことにしている。

`seq` という名の、何やらよくわからない変数が使われているが、これは次の理由で必要なものである。ひとつのプログラムの中には、`if` や `while` をたくさん書くのが普通なので、何度も分岐地点が出てくる。そのそれぞれの分岐において、別々のラベルを付けておかないと、いったいどこへ分岐すればいいのか決定できなくなってしまう。そのため、`seq` 変数を使って、全プログラム中でラベルを一意に決められるようにするのである。`seq` 変数の種は `CParseContext` で一元管理しておく（`getSeqId()` が呼ばれるたびに 1 増やしていけば一意に決まる）。

<sup>2</sup>更に言えば、0 なら偽と解釈し、それ以外なら真と解釈することになっている。

なぜこの生成コードで'<'に相当する分岐が実現できるのかについては、自分の頭で考えるべし。ここを自分の頭で理解することは極めて重要である<sup>3</sup>。

Condition.java

```
class ConditionLT extends CParseRule {
    private CParseRule left, right;
    private int seq;
    public ConditionLT(CParseContext pcx, CParseRule left) {
        this.left = left;
    }
    public void semanticCheck(CParseContext pcx) throws FatalErrorException {
        if (left != null && right != null) {
            left.semanticCheck(pcx);
            right.semanticCheck(pcx);
            if (!left.getCType().equals(right.getCType())) {
                pcx.fatalError(op, "左辺の型 [" + left.getCType().toString() + "] と右辺の型 ["
                    + right.getCType().toString() + "] が一致しないので比較できません");
            } else {
                this.setCType(CType.getCType(CType.T_bool));
                this.setConstant(true);
            }
        }
    }
    public void codeGen(CParseContext pcx) {
        PrintStream o = pcx.getIOContext().getOutputStream();
        o.println(";;; condition < (compare) starts");
        if (left != null && right != null) {
            left.codeGen(pcx);
            right.codeGen(pcx);
            int seq = pcx.getSeqId();
            o.println("\tMOV\t-(R6), R0\t; ConditionLT: 2数を取り出して、比べる");
            o.println("\tMOV\t-(R6), R1\t; ConditionLT:");
            o.println("\tMOV\t#0x0001, R2\t; ConditionLT: set true");
            o.println("\tCMP\tR0, R1\t; ConditionLT: R1<R0 = R1-R0<0");
            o.println("\tBRN\tLT" + seq + " ; ConditionLT:");
            o.println("\tCLR\tR2\t; ConditionLT: set false");
            o.println("LT" + seq + ":\tMOV\tR2, (R6)+\t; ConditionLT:");
        }
        o.println(";;;condition < (compare) completes");
    }
}
```

7. 上記の方法を参考にして、他の5比較演算子についてもコンパイルできる MiniC ver.07 を完成させよ。
8. 教員から指示されたテストケースに基づいてテストを行え。
9. テストを終えたら、生成されるコードが本当に正しいかどうか教員または TA に確認してもらうこと。

<sup>3</sup>細かいことを言えば、CMP 命令を実行したときにオーバーフローする可能性がある。その場合でも大小関係を正しく判定するには、ここに書いてあるコード以上の作業をしなければいけないのだが、それを考えるのはこの実験の範囲を超える。だから、このプログラムでよしとしよう。



## 実験 8 : if 文、while 文の導入 miniC ver.08

### 概 要

条件判定が書けるようになったので、if 文と while 文、加えて input 文と output 文を導入する。これでプログラミング言語として最低限の機能が網羅される。

### 8.1 構文定義

ここでは、if, while, input, output の 4 つの文が書けるようにする。ただし、その文法規則の BNF は示さないで、各自で考えて作ること。その際、if 文については特によく考えること。if 文には else が書けるように構文を決めなければならないが、このときいい加減に考えると、LL(1) ではない構文規則になる。なお、'{' と '}' で囲まれたブロック内に局所変数の宣言ができるようにする必要はない。

1. 上記の BNF 記述を書いて、教員または TA に確認してもらうこと。

### 8.2 字句解析部

構文定義によって新しく追加された字句が認識できるように、プログラムを修正する<sup>1</sup>。

2. miniCV07 プロジェクトをコピーして miniCV08 プロジェクトを作成せよ。本章のプログラミングは、miniCV08 プロジェクトで行うこと。
3. 上記のことができるように字句解析部を改造せよ。

### 8.3 構文解析部

導入した文を解析する構文解析部分を作成する。構文解析のプログラミング量は少々多い。

<sup>1</sup>'{' は TK\_LCUR、'}' は TK\_RCUR とでもしておくといよい。英語では、() は parenthesis、[] は (square) bracket、{} は brace あるいは curly bracket。



4. 前述の構文定義に従う入力ファイルを解析できるように、プログラムを変更せよ。
5. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。

## 8.4 意味解析部

実は、今回は意味解析の新たな機能を追加する必要はない。前章までで、必要なことはすべて終わっている。ただし、解析木の子節点への `semanticCheck()` 呼び出しを忘れないこと。

## 8.5 コード生成部

ここでは `StatementIf` に限って説明をする。`StatementWhile` も同様に考えればできるはずだ。

まずは `condition` 部にコード生成を依頼する。その結果はスタックに残っているはずだから、スタックトップの値を取り出して、偽の値だったときの分岐先ラベル名を生成してそこへ条件分岐する命令を生成しておく。次は `then` 部分の文についてコード生成を依頼する。戻ってきたら、(`else` 部分の文を実行してはいけないので) もうひとつラベル名を生成してそこへ無条件分岐する命令を生成する。その直後に、さっき作ったひとつ目のラベルをこの場所に置く。その次は `else` 部分の文についてコード生成を依頼する。戻ってきたら、先ほど作ったふたつめのラベル名をこの場所に置く<sup>2</sup>。


`output` 文と `input` 文は、SEP-3 のメモリマップト I/O 用アドレス `0xFFE0` 使ってコード生成する。このとき、どちらの文でも、代入文だと思ってコードを作ればよい(下図参照)。`output` の場合は、スタックトップの値を右辺の計算結果とみなし、`0xFFE0` (に割り当てられている変数) への代入文だと考える。`input` の場合は、キーワード `input` の後に代入文の左辺が書いてあり、`0xFFE0` (に割り当てられている変数) が右辺だと思えばよい。

input と output の意味

```
const int *led = &0xFFE0;
int i;
*led = 3;           // output 3; と同等
i = *led;           // input i; と同等
```

代入文のつもりでコード生成をするからには、両辺の型が一致していることを確認する意味解析が必要ではないかと考えるのは、ここまでの実験を真面目にやってきたことの証でもある。しかしながら、入出力文でそれを実施するのは「やりすぎ」である。なぜなら、入力される値の型を意識するのは人間だけであって、SEP にとっては意味がないからだ。言い換えると、整数値 3 を出力しようと、ポインタを表す 3 (番地の意味で) を出力しようと、LED には値 3 が表示されるだけだ。同様に、入力スイッチに 5 がセットしてあれば、それを整数型に代入するなら、人間は値 5 のつもりでセットしているわけだし、ポインタ型に代入するなら、人間は 5 番地を意味するつもりでセットしているわけだ。結局、出力値、入力値ともに、意味解析レベルでは型が決められないことを意味する。だから、ここで意味解析するのはやりすぎである。

<sup>2</sup>この方法だと、`else` 部がない `if` 文があったときに無駄な命令とラベルが作られるが、気にする必要はない。基本的な考え方を理解することが重要。

- 
6. MiniC ver.08 を完成させよ。
  7. 教員から指示されたテストケースに基づいてテストを行え。
  8. テストを終えたら、教員または TA に確認してもらうこと。

ここまでが終われば、最低限の機能を持ったプログラミング言語 miniC の第 1 版が完成である。ここまで完了すれば情報科学実験Ⅱの最低合格基準はクリアである。

ここから先はオプション課題であり、進捗程度によって成績が変わる。オプション課題は、原則としてそれぞれが独立しているので、どこから始めてもよい。時間のかかる課題もあれば、そうでない課題もある。



## 第II部

# オプション実験

第Ⅱ部では、少し高度な話題を扱う。

具体的には、次の項目を取り上げる。

- 関数が扱えるようにする（かなり大変なので3章にわたる）。
- 複雑な条件判定が書けるようにする。
- 入力ファイルの途中にエラーがあってもすぐに終わらずに、ファイルを最後までちゃんと読んで、できるだけたくさんのエラーを見つけてくれるようにする。
- 簡単な最適化をする。

関数をちゃんと扱えるようにするには、かなり頭を使うし作業量も多い。そのため3章に分割してある。これを順に実施していけば、完全にできるはずだ。

関数が扱えるようにするための3章は順に実施しないと完成しないが、その他の3章は完全に独立している。そのため、4つのうちのどこから始めても問題なく実施できる。

関数実現の3章を含む5章以上を完了したら「秀」、関数実現の3章をすべて完了したら「優」、どれでもいいから3章分を完了したら「良」とする。だから、関数実現の途中でちょっと詰まったら、先に他の（より容易な）章をやっ飛ばせば「良」にすることができる。その後で関数に戻ればよい。こういう順序で実施すれば、関数が全部終わったら「秀」は目の前だ。

## 実験9： 局所変数が使用可能な miniC

### 概 要

ここまでの miniC では、変数は大域変数のみが許されていた。この章では、局所変数を導入しよう。

### 9.1 構文定義

まずは、局所変数を使える構文を定義しよう。

```
program      ::= { declaraion } { declblock } EOF
declblock    ::= LCUR { declaration } { statement } RCUR    (注) LCUR='{', RCUR='}'
```

つまり、`'{'` と `'}'` で囲まれた範囲の中でのみ有効な変数群を用意する<sup>1</sup>。

### 9.2 字句解析部

新しく導入する字句はない。

### 9.3 構文解析部

前述の構文規則を解析するプログラムを作ろう。

### 9.4 意味解析部

ここでは、大域変数のほかに、局所変数を管理する必要が出てくる。第6.4節で触れておいたが、記号表(CSymbolTable クラス) には、局所変数を管理するための変数 local を用意しておいたので、これを使う。

今回の構文定義において、局所変数の有効範囲は(常識的に) `'{'` と `'}'` で囲まれた範囲であるから、変数 local に局所変数用の HashMap を登録するのは `'{'` が出現したときだし、削除するのは `'}'` が出現したときであることは問題なく理解できると思う。そのためのメソッド、例えば `setupLocalSymbolTable()` と `deleteLocalSymbolTable()` を、CSymbolTable クラスに実装しよう。これにともない、記号表の検索と登録のメソッド内部のアルゴリズムを修正する必要がある。検索は、局所変数用記号表をまず探し、見つからないときに大域変数用記号表を探すことになる。登録は、局所変数用記号表が存在すればそちらに、存在しなければ大域変数用記号表に登録する。もちろん識別子の二重宣言のチェックもする必要はあるが、局所変数の `i` が大域変数として登録済みであってもエラーとする必要はない。

このとき、CSymbolTableEntry の isGlobal 情報は、この登録メソッドの中で書き込んでやるのが、情報隠蔽の観点から考えてもよいことがわかるであろう(そのように修正せよ)。

<sup>1</sup>この declblock が、複数文をまとめて 1 文であるかのように扱う `'{ statement }'` と形が似ているからといって、if や while でも declblock が書けるようにするのは、よほどプログラミングに自信がある者以外はやめたほうがよい。局所変数の扱いをより複雑にし、真に実行できるコードを生成するのを難しくするだけだ(Cの言語仕様上は、そう書けるけれど...)。



図 9.1: フィボナッチプログラム実行時のスタックの様子

1. これまでに完成している最後のプロジェクトをコピーして、その次の番号のプロジェクトを作成せよ。本章のプログラミングは、新しいほうのプロジェクトで行うこと。
2. ここまでのことができるように、プログラムを変更せよ。
3. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。記号表を表示して、大域変数・局所変数の区別が付いているかどうか確認せよ。

## 9.5 コード生成部

第 I 部の必須実験で扱ってきた大域変数は、それが割り当てられるアドレスの管理をすべてアセンブラにまかせてしまっていた。そのため、コンパイラでは識別子名だけを扱えばよかった（例えば、`#var` という形で）。これは、大域変数が割り当てられたアドレスは、コンパイル済み SEP-3 プログラム実行時にはずっと固定されているからこそ可能なことである。

一方、局所変数はそうはいかない。次のプログラムを考えてみよう。おなじみのフィボナッチ数列計算プログラムだ。

```

fib.c
int fib(int n) {
    int answer;
    if (n == 1) {
        answer = 1;
    } else if (n == 2) {
        answer = 1;
    } else {
        answer = fib(n-1) + fib(n-2);
    }
    return answer;
}

```

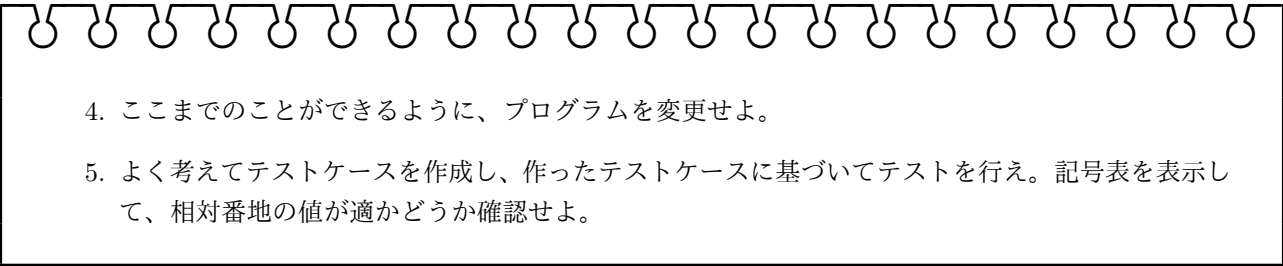
このプログラムを  $n=3$  で実行して  $n=1$  まで進行したときのスタックの様子は、「コンピュータ設計 I」や「機械語と計算機械」でやった通り図 9.1 の通りとなる。

「機械語と計算機械」では、「引数  $n$  が現在のスタックポインタ (SP) から見てどの位置にあるか」を問題にしたが、「関数が呼ばれるたびに新しく場所が確保される」局所変数の性質から、`answer` についてもこれと同じことが問題になるわけだ。

そこで、コード生成時に局所変数の存在位置を求めるためには、変数登録の時点で、スタックポインタからの相対位置を計算しておく必要がある。ただ、実はこれが結構大変な作業である。なぜなら、局所変数がひとつ追加されるということは、以前に記録しておいた変数の相対位置がすべて変わってしまうことを意味するからだ。ひとつの変数を追加するたびに記号表を総ざらいして、新たに登録し直すなどということは誰もしたくない。

そこで考えられたのが、「フレームポインタ」という概念である。一般に、「JSR 命令を実行して関数にやってきたときのスタックポインタの位置」のことを言い<sup>2</sup>、これを特殊目的のレジスタにコピーしておくのである。CPUの中には、ハードウェアが（サブルーチンコール命令の実行サイクルの中で）やってくれるものもあるが、残念ながら SEP-3 にはそのような目的のレジスタは用意されていないし、そのようなハードウェアもない。しかしながら、この実験のコード生成で利用しているレジスタは、ここまで R0 から R2 までの 3 本だけである（確認せよ。もしこれ以上に使っているとしたら、相当無駄なことをしていることになる）。余っている R4 をフレームポインタとして利用することにしよう。

この決定により、局所変数は「フレームポインタからの相対番地」によって位置決めができることになる。スタックポインタからの相対番地だと負の方向になっていたものが、フレームポインタからの相対位置なら正の方向になるわけだ。そして、フレームポインタは関数実行中に移動しないので、記号表に書き込む相対番地も値を変更する必要がなくなり、総ざらいして書き直す必要もなくなる。

- 
4. ここまでのことができるように、プログラムを変更せよ。
  5. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。記号表を表示して、相対番地の値が適かどうか確認せよ。

ここまでできれば、あとは局所変数にアクセスする方法を考えるだけである。変数アクセスはすべて `Ident` クラスで行っているから、ここを書き換えるだけで済む。やるべきことは、(1) その変数が大域変数なのか局所変数なのかをまず判定する。大域変数なら今まで通り。(2) 局所変数だった場合は、フレームポインタの値と相対位置を足して、その変数用に割り当てられた番地を計算するようなコードを生成する。この計算はコンパイラ自身がやってはいけない。あくまでも、生成するコードを実行したときに、コード自身が計算するように仕向けなければならない。コンパイル時にコンパイラが計算してしまうということは、そのコードを実行するときにはアドレスが固定されてしまっていることを意味する。それでは局所変数ではなく、大域変数の扱いになってしまう。この違いについてよく理解すること。


あとひとつ重要なことがある。局所変数用の領域を、実際にスタック上に確保する仕事である。

が、その前にまず、旧いフレームポインタの値を保存しておく必要があることに気付いて欲しい。つまり、局所変数の有効範囲から出るとき（関数から復帰するとき）、以前のフレームポインタに戻してやらねばならないということである。したがって、局所変数用の記号表を作るとき、削除するときにあわせて、フレームポインタの値を（スタックに）保存し復旧してやらねばならないのである。

さて、本論の局所変数用領域確保について。`DeclBlock` クラスの宣言部の解析が終わった時点（文の解析を始める前の時点）で、局所変数用記号表を調べれば、どれだけの局所変数領域が必要なかがわかる。この数値を持ってきてスタックポインタに加えてやることで、一気に領域の確保ができる。ただし、局所変数用記号表はコード生成時には消えてなくなっている（6.4 節を思い出せ）。例によって、構文解析時に `private` 変数に必要領域量を蓄え、コード生成時に参照するように作ることが必要だ。

<sup>2</sup>図 9.1 では、「戻り番地」のすぐ下のところ。





6. 以上のことができるように、プログラムを変更せよ。

7. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。生成されたコードを確認して、局所変数が正しくアクセスできていることを確かめよ。

## 実験 10： 引数なし関数が使用可能な miniC

### 概 要

本章を始めるには、実験 9 が完了している必要がある。また、かなりのプログラミング量がある。

関数が定義でき、呼び出すことができるようにしよう。その際、関数は再帰的呼び出しが可能であるようにしよう。

### 10.1 構文定義

まずは、関数の定義ができる構文を定義しよう。ちょっと言葉を変えると、宣言ブロック (declblock) に、名前を付けることができる...ということでもある。

```

program      ::= { declaraion } { function } EOF
declaration  ::= intDecl | constDecl | voidDecl
voidDecl     ::= VOID IDENT LPAR RPAR { COMMA IDENT LPAR RPAR } SEMI
declItem     ::= [ MULT ] IDENT [ LBRA NUMBER RBRA | LPAR RPAR ]
function     ::= FUNC ( INT [ MULT ] | VOID ) IDENT LPAR RPAR declblock
statement    ::= (長いので省略) | statementCall | statementReturn
statementCall ::= CALL ident LPAR RPAR SEMI
statementReturn ::= RETURN [ expression ] SEMI
variable     ::= ident [ array | call ]
call         ::= LPAR RPAR

```

関数 (function) は、void 型 int 型 int\* 型の 3 種を許している。今のところ、引数を扱うことはしない (これは次章のテーマ)。が、返却値は扱える (statementReturn)。「大域変数で情報を渡すと、何か計算してくれて返却値を戻してくれる」ような何ものかを想像してくればよい。

そして、関数を呼び出すための構文として関数呼び出し文 (statementCall) が追加されるとともに、式の中でも使えるように variable が変更されている。

### 10.2 字句解析部

新しい型 void と、return 文用キーワードが追加された。関数であることを示す func と呼び出しを表す call も追加されているが、これは、LL(1) であることを守ろうとすると、導入するほうが楽だからそうしてある。なければならないで LL(1) 文法を守るようにすることはできるが、意味解析やコード生成の場面でかなり混乱と思うので、今回は避けている。この 2 つのキーワードなしで頑張りたい学生は、やってみるとよい。

1. これまでに完成している最後のプロジェクトをコピーして、その次の番号のプロジェクトを作成せよ。本章のプログラミングは、新しいほうのプロジェクトで行うこと。
2. 前述の構文定義に従う入力ファイルを解析できるように、プログラムを変更せよ。
3. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。

### 10.3 構文解析部

前述の構文規則を解析するプログラムを作ろう。

4. 前述の構文定義に従う入力ファイルを解析できるように、プログラムを変更せよ。
5. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。

### 10.4 意味解析部

新しい型 `void` に関して、型計算（型推論）アルゴリズムを変更する必要がある。ただし、実際には関数に関わる場所だけが問題なので、通常の演算子（加減乗除、比較演算など）について `void` を扱えるようにする必要はない。関数呼び出しに関わる節点だけで問題にすればよい。

さて、今回の構文の変更によって、意味解析においてかなり重大な課題が生じているのだが、それが何かわかるだろうか？

答えは、`variable` の扱いである。何のことやら分からない学生は、第 4.1 節をもう一度よく読んで欲しい。

`variable` は、「番地」を意味する節点であった。ここに、関数呼び出し記法を加えるというのが、本章の変更点である。関数呼び出しの結果は「値」でしかないのに、これをどう「番地」にしてやるか...というのが重大な課題なわけである。「番地」から「値」にするには、「番地」を参照する（読み出す）ことでそこに入っている「値」を取り出してやればよかった。しかし、もともと「値」しかないものを、どうやって「番地」の情報に変換してやればいいのかは、大問題である。

解決法は 2 つある。

ひとつは、そういう面倒なことは考えるのをやめて、関数呼び出しを「値」しか扱わない節点（構文規則）に追い出すという方法である。`unsignedFactor` の節点にさかのぼって（第 4.1 節を参照）、ここに関数呼び出しを加えてやるのである。その場合は、`LL(1)` であることを保証するために、関数呼び出しであることを示す字句を新たに導入して（`primary` の `first` 集合と競合しないように）制御しなくてはならない。例えば、`a + @f() + 1` とか `a + call f() + 1` とかいった記法にする。

もうひとつは、関数の返却値を入れておくための「プログラマには見えない局所変数」を内部的に作り出し、関数からの返却値をそこに代入したうえで、その変数の番地を与えるというやり方である。

どちらの方法を採用しても良い。なお、後者を選んだ場合には「プログラマには見えない局所変数」の名前を生成する必要があるけれど、それは第 7.5 節で説明した「ラベル名を一意に作り出す方法」を準用すればよい。

これ以外にも、まだいくつか考えなければならないことがある。

相互再帰的関数の組（関数 `f` が関数 `g` を呼び、関数 `g` が関数 `f` を呼ぶようなもの）が出てくると、困ったことがある。つまり、関数 `f` が関数 `g` よりも先に定義されていると、関数 `f` の中で関数 `g` が呼ばれているときに、その型

(返却値の型) をどうやって決めたらいいの?ということである。この場合、C 言語で言うところの「プロトタイプ宣言」がどうしても必要になる。上記の構文規則では、その宣言を `declItem` の中で行えるようにしてある。

相互再帰関数の組.c

```
func int f() {
    int x;
    x = g();    // どうやって意味解析するの、これ? g の定義は 4 行先に初めて出てくるまで
    return 1;   // わからんのだけど...
}

func void g() {
    int y;
    y = f();
}
```

これを実現するには、関数の宣言 (`declItem` の中で行う) と関数の定義 (`function` の中で行う) とで、識別子を記号表に登録する際に別扱いすることが必要になってくる。つまり、宣言のときの情報は仮登録としておき、定義が出てきたときに仮登録と矛盾していないかを確認するのである。

先ほど話題にした `variable` について、「プログラマには見えない局所変数」を導入して解決した場合に限って生じる問題がある。この場合、`variable` は代入文の左辺に書いてよいことになっている (実験 5)。ということは、代入文を `f() = 3;` と書いても文法的には正しい (が、意味を成さない)。これを検出するには、「プログラマには見えない局所変数」を“定数”として扱えばよい。「定数には代入できない」ことを代入の実現 (実験 5) で実装したので、それをここで利用するのである。

最後にもうひとつ。`return` 文の後ろに式を書いてある場合は、それが関数の定義に示された型と一致しているかどうかを検査する必要がある。また、式がない場合には、その関数が `void` 型と定義されているかどうかをチェックする。


6. 前述の意味解析を実現せよ。

7. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。

## 10.5 コード生成部

それぞれの関数の最初の命令のところに、関数名と同じラベルを生成しておく。プログラム中で関数を呼び出すところでは、“JSR その関数名” 命令を生成してやればよい。

`return` 文については、例えば、後ろの式の計算結果を `R0` に残すと決め (他の場所に決めても、もちろん可)、`RET` 命令を生成すればいい。ただし、問題がひとつある。それは、単純に `RET` 命令を実行してはいけないということにある。なぜなら、スタック上には局所変数領域が取られているので、それを全部削除しなければ戻り番地が出てこないからである (図 9.1, p.48)。これを避けるには、`return` 文に対しては“`JMP RET_関数名`”のような命令を生成しておくことにし、関数の最後のところへラベル“`RET_関数名`”を置き、ここから局所変数領域を捨てる命令と `RET` 命令を生成する。このとき、「局所変数領域」の先頭番地はフレームポインタが抑えている (第 9.5 節) ので、たとえ「プログラマには見えない局所変数」を使ったために局所変数の領域数がわからなくなっていたとしても、“`MOV R4, R6`”の 1 命令で確実に正しい位置までスタックポインタを戻すことができる。その後で `RET` 命令を実行すれば (旧フレームポインタの値を復旧させるのが先かもしれないが)、これで呼び出し元に正しく戻れる。



8. コード生成を実現せよ。

9. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。

## 実験 11： 引数付き関数が使用可能な miniC

### 概 要

本章を始めるには、実験 10 が完了している必要がある。

関数に引数を渡すことができるようにしよう。これで、手続き型プログラミング言語の機能は、ほぼ網羅したことになる。

### 11.1 構文定義

まずは、引数を扱うための構文を定義しよう。

```
function      ::= FUNC ( INT [ MULT ] | VOID ) IDENT LPAR [ argList ] RPAR declblock
arglist       ::= argItem { COMMA argItem }
argItem       ::= INT [ MULT ] IDENT [ LBRA RBRA ]
statementCall ::= CALL ident LPAR [ expression { COMMA expression } ] RPAR SEMI
call          ::= LPAR [ expressoin { COMMA expression } ] RPAR

voidDecl      ::= VOID IDENT LPAR [ typelist ] RPAR
                { COMMA IDENT LPAR [ typeList ] RPAR } SEMI
declItem      ::= [ MULT ] IDENT [ LBRA NUMBER RBRA | LPAR [ typeList ] RPAR ]

typeList      ::= typeItem { COMMA typeItem }
typeItem      ::= INT [ MULT ] [ LBRA RBRA ]
```

`argItem` を `declItem` と別に定義したのは、(1) 前者には関数宣言が来ることはない、(2) 前者では配列の大きさは必要ない<sup>1</sup>... の 2 つの理由による。また、さらに `typeList` が追加されているのは、いわゆるプロトタイプ宣言における型の指定ができるようにするためである。ここには、識別子を指定する必要がない。

### 11.2 字句解析部

新しく導入する字句はない。

### 11.3 構文解析部

前述の構文規則を解析するプログラムを作ろう。

<sup>1</sup> 引数として渡すのは、配列の全体ではなく、配列の先頭アドレス（これは定数）だけである。

1. これまでに完成している最後のプロジェクトをコピーして、その次の番号のプロジェクトを作成せよ。本章のプログラミングは、新しいほうのプロジェクトで行うこと。
2. 前述の構文定義に従う入力ファイルを解析できるように、プログラムを変更せよ。
3. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。

## 11.4 意味解析部

今回新たに付け加えるべきことは、関数を呼び出す側 (`variable` や `statementCall` において記述されている引数；これを実引数と呼ぶ) と、呼び出される側 (`function` において記述されている引数；これを仮引数と呼ぶ) とで、引数の数と、それぞれの位置における引数の型が一致している、かどうか... という点である。

これを実現するには、関数名を記号表に登録する際に、引数に関する情報も合わせて登録しておく必要がある。ただし、現状では記号表エントリ (`CSymbolTableEntry`) にそれ用の場所がないので、引数情報を入れておくクラス変数をひとつ追加すると良い。

「機械語と計算機械」のフィボナッチ数計算プログラムでやった通り、引数はスタックに積むことで関数に渡す。このとき、引数を左から順に積む方法と右から順に積む方法がある<sup>2</sup>。どちらのほうがよいか、以下を読んでよく考えること<sup>3</sup>。

まず最初に考えるべきことは、仮引数は大域変数なのか、局所変数なのかという点である。明らかに当該関数実行中にのみ有効な局所変数なのだから、局所変数用記号表を用意したり削除したりする位置 (実験 9) を見直す必要があることに気が付いて欲しい。

また、これが局所変数だということは、フレームポインタからの相対位置はマイナスになるということでもある。単純に局所変数として記号表に登録していくと (ここまでのプログラムを普通に作ってきた学生なら、通常は 0 から順に割り当てははずだから)、相対位置が常に正の値になってしまうはずだ。これを (解析途中の) 適切な位置で<sup>4</sup> 負の値に調整してやる必要がある。このとき、JSR 命令が勝手にスタックに積む戻り番地と、旧フレームポインタの保存位置 (実験 9) があることを忘れてはならない。絵を描けば、どういう状況になるか一発でわかるはずだ (図 11.1)。

さて、ここでひとつ微妙な問題がある。後述するクイックソートの呼び出しプログラム (`qsort.c`) を見て欲しい。そこでは、引数として「配列 `a[]` の先頭アドレス」を渡したいのだが、ここに少々困難があるのだ。

ここまで普通に考えてプログラミングしてきたとすると、`int[]` 型のものに演算子 `&` を付けることはできないように実装しているだろう (実験 4 の意味解析)。かと言って、単純に `a` の形で渡そうとすると、それはダメだと言われるか (実験 4 の意味解析)、`a[0]` が保持する整数を一つだけ渡すことになってしまうだろう (実験 4 のコード生成)。それなら... と、「配列 `a[]` の先頭アドレスは 0 番目の要素のアドレスでもあるから、`&a[0]` の形で渡そうとすると、今度は、これを受け取る側ではポインタ型をひとつしか受け取れないことになる。つまり、関数 `qsort()` 内では、受け取ったものを配列としてアクセスできない (実験 11 の意味解析)。

これを解消する一番よい (簡単な) 方法は、「`int[]` 型のものに演算子 `&` を付けることを許し、にもかかわらず、その型は `int[]` のままに据え置く<sup>5</sup>」ことだ。これは `factorAmp` の意味解析部分を少々変更することで簡単に実現できる。

<sup>2</sup> “`printf` のような可変個引数関数を実現するにはどうすれば良いか” というレポート課題に対して、“引数の個数を数えて最後に積む” 以外に、“引数を右から順に積む” というエレガントな解があるのだが、これに気づいている学生は少なかった。

<sup>3</sup> どちらを採用してもプログラミングできる。どちらのほうが考えやすいか、プログラミングしやすいかは人による。

<sup>4</sup> どの節点で行うべきか、また `parse()` で行うべきか、`semanticCheck()` で行うべきかは、よく考えて決めること。

<sup>5</sup> 普通に考えれば、その型は「配列へのポインタ」であり、これは、これまで頻繁に出てきた「ポインタの配列」とは異なる型だ。

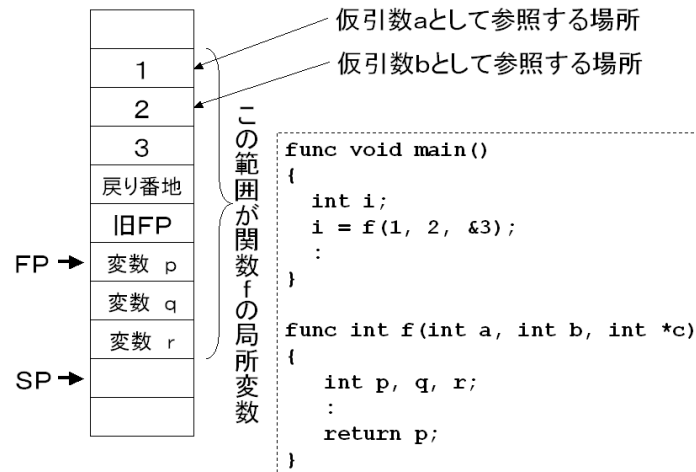


図 11.1: 引数付き関数実行時のスタックの様子。これは引数を左から順に積んだ場合の図である。FP はフレームポインタを意味する。

## 11.5 コード生成部

コード生成部で行うことは、(1) 引数をスタックに積むこと (図 11.1 参照)、そして (2) 関数から戻ってきたら積んだ分だけ引数を捨てることである。(1) については、式の計算コードを生成しておくだけで、結果として自然に引数がスタックに積まれることになるので、何も面倒なことはない。(2) を忘れずに行えば、それで OK だ。

1. 以上のことが実現できるように、プログラムを変更せよ。
2. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。
3. SEP ボードあるいはシミュレータ上で動作確認せよ。swap 関数を用いて要素を交換するクイックソートを実装し、コンパイル、アセンブルして実行してみると良い。これで、何ができていて何ができていないかが明確になる。

qsort.c

```
func void swap(int *a, int *b) {
    int t;
    t = *a; *a = *b; *b = t;
}

func void qsort(int a[], int start, int end) {
    // 配列 a の a[start] から a[end] までクイックソートで整列する
}

func void main() {
    int a[10];
    // a に初期値を設定する
    qsort(a, 0, 9); // C では、こう書けば配列 a の先頭アドレスが渡せる
    // だが、ここでは qsort(&a, 0, 9) と書かないと、正しく動くコードにならないかもしれない
    // a を出力する
}
```





## 実験 12： 複雑な条件判定が可能な miniC

### 概 要

実験 7 で導入した条件判定式は、ごく単純なものであった。本章では、もう少し複雑な条件が書けるように拡張しよう。具体的には、not、and、or が書けるようにする。

### 12.1 構文定義

C では、not、and、or をそれぞれ '!', '&&', '||' で表現している。そしてこれら（と大小比較の 6 演算子）との間での優先順位は、式の中で使う '+', '-', '\*', '/' の関係とよく似ている。したがって、式の BNF を参考にして条件判定式の BNF を作り、同じようにプログラミングすればよいだろう。

1. 条件判定式の BNF 記述を書いて、教員または TA に確認してもらうこと。なお、どうしても LL(1) に収まらない場合も出てくると思う。この場合、LL(1) にこだわると、文法規則を大幅改訂しなければならない。そうになってしまう場合は、LL(1) であることにこだわる必要はないので、ごく自然な記述ができるように決めればよい（オプション実験だし、普通にプログラミングしておく限り、動作が意図通りにならないことも起きない）。

### 12.2 字句解析部

新しく導入した字句を認識できるように、プログラムを変更する。

2. これまでに完成している最後のプロジェクトをコピーして、その次の番号のプロジェクトを作成せよ。本章のプログラミングは、新しいほうのプロジェクトで行うこと。
3. 上記のことができるように字句解析部を改造せよ。

### 12.3 構文解析部

導入した条件判定式を解析する構文解析部分を作成する。

表 12.1: 真偽値の決め方と、使う SEP-3 命令

	論理否定 (not)	論理積 (and)	論理和 (or)
1 が真、0 が偽	XOR #0x0001	AND	OR
1 が真、-1 が偽	XOR #0xFFFE	OR	AND

## 12.4 意味解析部

ここは式の意味解析と同様にすればよい。

ここでは論理積 (and) を例にして説明する。and 節点の下につながる左右の解析木は、当然ながらどちらもブール値を型として持つはずである。したがって、この and 節点では左右の解析木が T\_bool と T\_bool であることを確認し、よければ自身も T\_bool に設定するし、そうでなければ意味エラーを発生させなければならない。構文解析がエラーなく終了していれば、ここで意味エラーが発生するはずはない。しかし、エラー回復 (実験 13) を実現すると、構文解析が中途半端に終わってしまうこともあるので、サボらずにチェックしておくべきである。

4. 前述の構文定義に従う入力ファイルを解析できるように、プログラムを変更せよ。また、意味解析を追加せよ。
5. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。

## 12.5 コード生成部

ブール値の計算をするだけなので、難しくはないはずだ。実験 7 において、真偽の値をどのように決めたかによって、生成するコードがちがってくる (表 12.1)。真偽値に 1 と 0 を使っているとき (上段) については 1 年生のとき以来何度も何度も目にしているはずなので皆さんもよく理解しているはずだが、真偽値に 1 と -1 を使うとき (下段) の結果は、直感とは正反対の常識はずれのように感じて目を疑うことだろう。なぜこれでうまくいくのか各自よく考えること。よい頭の体操になる。

6. 複雑な条件判定ができる miniC を完成させよ。
7. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。
8. テストを終えたら、教員または TA に確認してもらうこと。

## 実験 13： エラー回復が可能な miniC

### 概 要

ここまでの miniC は、文法エラーや意味解析エラーがあると単にエラーメッセージを表示してコンパイルを中止するだけで、誤りを適切に解釈して処理してくれるわけではなかった。しかし、それでは全然実用にはならない。現実に使われている実用的コンパイラはすべて、エラーがあっても可能な限り作業を続行し、入力ファイルに含まれるエラーを最大限見つけ出して通知してくれる。

miniC も、ちょっと気の利いたまともなエラー処理ができるコンパイラに改造しよう。

### 13.1 エラーのレベル

コンパイルエラーには、そのエラーの致命的度合いによっていくつかのレベルが設定できる。

1. 意味解析上のエラー。例えば、ポインタ型どうしを乗算することが発見されたなど。このときは、多くの場合、何も問題なくコード生成が可能である。そのコードを実行してみたとき、意味のない計算が行われてプログラムの意図通りに動かないというだけのことだ。したがって、コンパイラは少しだけ文句を言って、コンパイル作業自体は続行することができる。
2. 文末の';' や、'if' 直後の '(' などの書き忘れ。これは明らかな人間のミスであって、構文定義を知っているコンパイラとしては、そこに何を補えばいいのかを間違いなく確定できる。このときは、コンパイラが適切な字句を補って何事もなかったかのように（ただし、少しだけ文句を言って）コンパイル作業を続行すればよい。
3. 式の解析途中で、識別子が来るべきところにそれが来ないとか、'+' か '-' のどちらかが来なければならないはずなのにどちらも来ないとかいったとき。このときコンパイラとしては、「適切な字句」を補ってコンパイルを続行することは不可能である。どの識別子が来るのが正しいのか、どちらの記号を補えばいいのかについて、「確実に」決定する手段は何もないからである。この場合コンパイラにできることは、構文解析が再開できる地点（例えば、何かの終わりを示す';' や']' などの字句、'if' などのキーワードを手がかりにする）まで字句を読み飛ばし—つまり、エラーが起きたあたりには何も書いてなかったことにして—、再開できるところを見つけてそこから再開することだ。
4. 致命的なエラー。入力ファイルが大きくて構文木が大きくなりすぎ、コンパイラが使えるメモリ領域を使い果たしてしまったなど。コンパイルを続行すること自体が不可能な場合が該当する。

これらを注意深く観察すれば、3つのレベルがあることがわかる。コンパイル続行不可能な「致命的なエラー」、字句読み飛ばしによって「回復できるエラー」、そして「文句を言うだけで事足りるエラー」である。

第 I 部の miniC はこれらの区別をせず、すべて「致命的エラー」の扱いであったことに気付くだろう。本章では、これを区別して、立ち直れる場合には立ち直るようにプログラミングすることを目指す。

### 13.2 各エラーレベルの実装方法

ここでは、各エラーレベルに対してどうやってプログラミングをしていくかについて解説する。

### 13.2.1 致命的エラー

「致命的エラー」の実装については、既に ver.00 からプログラムを読んでいるので理解していると思うが<sup>1</sup>、簡単に復習しておく。

`lang` パッケージ内に `FatalErrorException` クラスが用意されていて、この例外が `ParseContext.fatalError` メソッドにおいて投げられる。つまり、構文解析プログラム (`lang.c.parse` パッケージ内のクラスで構成される) から `fatalError` メソッドが呼ばれると、`FatalErrorException` が投げられることになる。そしてこの例外がどこで受け取られるかという点、`lang.c.MinCompiler` の `main` メソッドということになっている。よって、`fatalError` メソッドを呼んだら、エラーメッセージを出力して (例外を投げて受け取ることで) 直ちに `main` メソッドに戻って終了する... という仕掛けになっている。

よって、致命的エラーが発生したら、これをそのまま利用させてもらえばよい。つまり、`ParseContext.fatalError` メソッドを呼び出すようにすればよい。

### 13.2.2 文句を言うだけで事足りるエラー

`ParseContext.fatalError` メソッドと同じインタフェースのメソッド、例えば `ParseContext.warning` を作って、例外を投げずに終了 (`return`) するように実装すればよい。そうすれば、呼び出し元にちゃんと戻ってきてくれる。戻ってきた後、必要なら「字句を補う」と同等の作業をプログラムで行えばよい。

### 13.2.3 回復できるエラー

これの実装が一番難しい。いくつかやり方はあるが、一番すっきりしてお勧めの方法は次の通りである。

1. まずは、`lang.FatalErrorException` を継承して `lang.RecoverableErrorException` を作る。

```
RecoverableErrorException.java
public class RecoverableErrorException extends FatalErrorException { }
```

2. `ParseContext.fatalError` メソッドをコピーして `recoverableError` メソッドをつくり、最後の行で `RecoverableErrorException` を投げるように変更する。

```
ParseContext.java
public void recoverableError(final String s) throws RecoverableErrorException {
    error(s);
    throw new RecoverableErrorException();
}
```

3. 構文解析部 (`lang.c.parse` パッケージ内のクラス群) では、`try...catch` をうまく使ってロジックを作る。このとき `catch` するのは `RecoverableErrorException` であって、そのスーパークラスである `fatalErrorException` のほうではないことに注意。 `fatalErrorException` は捕捉せずに素通りさせて、`main` まで届くようにする (それをするために継承したのだ)。 `block ( '{' と '}' で囲まれた文の並び )` の解析部分を例に挙げると、こんな具合だ。

<sup>1</sup>本テキストではここまで一切解説していないけれど...

Block.java

```
public void parse(CParseContext pcx) throws FatalErrorException {
    // ここにやってくる時は、必ず isFirst() が満たされている
    CTokenizer ct = pcx.getTokenizer();
    CToken tk = ct.getNextToken(pcx);          // '{' を読む
    statements = new ArrayList<CParseNode>();
    while (Statement.isFirst(tk)) {
        try {
            // 文の途中で構文エラーになるかもしれないので...
            CParseRule stmt = new Statement(pcx);
            stmt.parse(pcx);
            statements.add(stmt);
        } catch (RecoverableErrorException e) {
            // そのときにはここで例外を捕まえて、';' か '}' が出るまで読み飛ばして回復する（立ち直す）
            ct.skipTo(pcx, CToken.TK_SEMI, CToken.TK_RCUR);
            tk = ct.getNextToken(pcx);
        }
        tk = ct.getCurrentToken(pcx);
    }
    if (tk.getType() == CToken.TK_RCUR) {
        tk = ct.getNextToken(pcx);
    } else {
        pcx.warning(tk, "}"が閉じていませんので補いました");
    }
}
```

1. これまでに完成している最後のプロジェクトをコピーして、その次の番号のプロジェクトを作成せよ。本章のプログラミングは、新しいほうのプロジェクトで行うこと。
2. 上記のことができるようにエラー処理を実装せよ。
3. 正しい構文規則にしたがって書いたプログラムを用意し、それをあちこち削ったり、余分なものを書いたりして、入力してみよ。思った通りに立ち直すコンパイラになっているか？ 特に、自分が良くやるミスをうまくカバーしてくれるコンパイラになっているか？
4. テストを終えたら、教員または TA に確認してもらうこと。



## 実験 14： 多少の最適化が可能な miniC

### 概 要

最適化について、少しだけ考えてみよう。ここでは、3 年生でも十分に実装できる、比較的簡単な（でも効果は抜群の）最適化技法について実現してみる。

### 14.1 レジスタ使用の最適化

実験 0 以来ここまで、式の計算その他において、計算の途中結果はすべてスタックに置いてきた。そのため、生成されたコードを読むと push と pop が繰り返し行われ、冗長な部分がひどく多いなと感じていることと思う。

そこで、その冗長さを少し緩和することを考えよう。途中結果をすべてスタックに積むのではなく、レジスタを有効に使うのである。レジスタ割り当て技法の難しい手法はいろいろあるのだが、3 年生でも十分に理解できる簡単な方法を、ここでは扱う。

ここまでの方法では、演算子のオペランドはすべてスタックに積まれていることを仮定していた。実は、この決定が冗長さを生み出す癌であった。幸いなことに、この実験で定義したプログラミング言語は 1 ワードの大きさの型しか扱わないので、オペランドは常にレジスタ上に保持できる。そこで、「(構文規則の) 非終端記号のコード生成が終わったときには、その計算結果を常に R0 に保持している」というように仮定を変えることにしよう。すると、二項演算子の場合—つまり、“式 1 演算子 式 2” のような場合—には、次のようにコードを生成してやればよいことになる。

- まず、式 1 のコード生成をする。
- 終わって戻ってきたときには、答えが R0 に残っている（ようなコードが生成されている）。この値（つまり R0）を（式 2 の計算途中に書き換えられても困らないように）スタックに積む。
- 式 2 のコード生成をする。
- これまた終わって戻ってきたときには、答えが R0 に残っている（ようなコードが生成されている）。スタックから式 1 の答えを取り出して R0 と演算し、答えを R0 に残す。

これまでは帰りがけにだけコード生成を行っていたが、左子節点（式 1 の節点）から戻ってきて右子節点（式 2 の節点）に移るときにもコード生成をする... ように変えるということである。たったこれだけのことだが、生成されるコード量は劇的に削減され、効果は絶大である。試してみて欲しい。

1. これまでに完成している最後のプロジェクトをコピーして、その次の番号のプロジェクトを作成せよ。本章のプログラミングは、新しいほうのプロジェクトで行うこと。
2. 上記の決定に基づいて、コード生成部を全部見直してみよ。どのタイミングで push と pop をすればよいか？
3. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。生成されたコードは、目的の動作をするか？（スタックの扱い方がおかしくないか？）



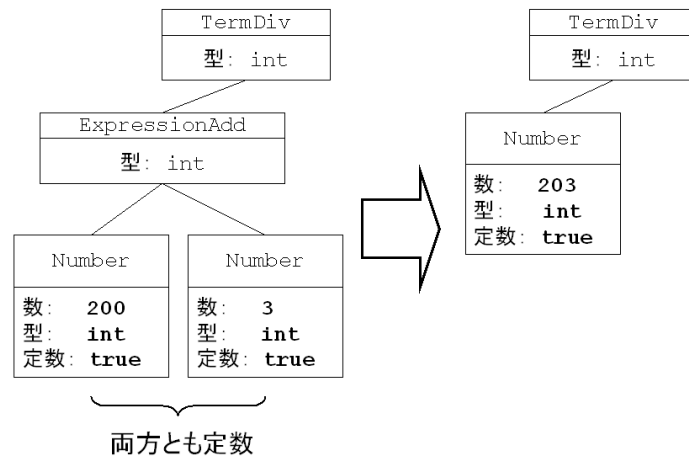


図 14.1: 定数の事前計算例

## 14.2 定数の事前計算による最適化

本実験では、(構文) 解析木の各節点には、それが定数を表すかどうかを示す情報が格納してあった。この情報を用いることで、コンパイル時に、定数の事前計算を(ある程度まで)することができる<sup>1</sup>。例えば、(1) $i + 3 * 2$  を  $i + 6$  と見てコンパイルしたり、(2)200 番地に割り当てられた大域配列  $a$  に対して  $a[3]$  を、「200 をもってきて、3 をもってきて、足して、アクセスする」コードのかわりに、一発で「203 をアクセスする」コードを生成しようというのである。

この作業は、解析木を深さ優先で走査して上の節点に戻るときに実施することができる。例えば足し算の節点を例に採って説明しよう(図 14.1)。

`ExpressionAdd` の節点において、両辺の節点が定数であるかどうかを確認する。左辺は `Number`、右辺もまた `Number` だから、加算結果も(つまり自分自身も)定数になれることがわかる<sup>2</sup>。そこで、左辺の定数値を取り出し、右辺からも定数値を取り出してそれらを足し(なぜなら自分自身が `ExpressionAdd` だから)、新たな `Number` 節点を作って値 203 を格納して、呼び出し元にこれを返すのである。呼び出し元(図では `TermDiv`)では、戻ってきた値(節点)を自分自身が管理する新たな節点にする。

これを実現するには、2つの方法がある。

- 手っ取り早く片付けるなら、`semanticCheck()` メソッドが(現在は `void` だが) `CParserRule` を返すように変更し、意味解析と一っしょに実施してしまう。
- 真面目に設計するなら、`Compiler` を次のように変えて、

Compiler.java

```
public interface Compiler<Pctx, Tree> {
    public abstract void semanticCheck(Pctx pcx) throws FatalErrorException;
    public abstract void codeGen(Pctx pcx) throws FatalErrorException;
    public abstract Tree optimize(Pctx pcx) throws FatalErrorException;
}
```

さらに、`CParserRule` を次のように変えて、

<sup>1</sup>ここに示す単純な方法だと、「 $3 + i + 5$ 」のように、定数と定数の間に変数が入っているような場合に、これを「 $i + 8$ 」へと変換するのは簡単ではない。

<sup>2</sup>なお、定数だと印が付いていたとしても、実行時に定数になるというだけでコンパイル時にはその値が決められないことがある。例えば、局所変数の割り当てアドレスを求める `&localvar` のようなものがそうである。このようなものはちゃんと検出しなければならない(局所変数が大域変数かの情報は、`CSymbolTableEntry` に含めてあったから、これを参照すればできる)。

CParseRule.java

```
public abstract class CParseRule extends ParseRule<CParseContext>
    implements lang.Compiler<CParseContext, CParseRule>, LL1<CToken> {
```

そのうえで各節点に `optimize()` メソッドを記述する。もちろん、`MiniCompiler` も

MiniCompiler.java

```
if (pcx.hasNoError()) parseTree.semanticCheck(pcx);    // 意味解析
if (pcx.hasNoError()) parseTree.optimize(pcx);         // 最適化      <== ここ！
if (pcx.hasNoError()) parseTree.codeGen(pcx);          // コード生成
```

と、最適化のメソッドを呼び出すようにする必要がある。

また、大域変数のアドレス割り当てをアセンブラにまかせておいてはいけない。コンパイラ自身で管理する必要がある。

1. 以上のことが実現できるように、プログラムを変更せよ。
2. よく考えてテストケースを作成し、作ったテストケースに基づいてテストを行え。