

プログラム言語論 第3週

亀山幸義, 2009/04/27

1 今日の目標

以下のことを理解すること。

- プログラム (の構文) は、木である。
- (ALGOL 型のプログラム言語における) ブロック構造。
- スコープ、大域変数、局所変数。
- 静的束縛、動的束縛。

2 MiniC 言語の構文

前回までの MiniC は、C 言語の一種とは言えないほど小さく (関数定義すらなかった)、かつ、C 言語とは少し異なる構文を採用していた。今週は、演習を行うこともあり、なるべく混乱をきたさないよう、より C 言語に近い言語を採用し、これを MiniC と呼ぶことにする。(前回までの、関数定義しない言語のことを呼ぶ必要があるときは、MicroC と呼ぶ。)

拡張後の MiniC の構文を、表 1 に記載する。(なお、コメントは一切書けない。)

MiniC のプログラム例を以下にあげる。

```
int main () {
    int zero;
    int minus_one;
    int sum;
    int n;
    zero = 0;
    minus_one = -1;
    sum = 0;
    n = 10;
    while (n > zero) {
        sum = sum + n;
        n = n + minus_one;
    };
    print sum;
}
```

2 項演算子の `-` がないので、「1 を引く」という操作のかわりに `-1` を加えている。

また、型として `int` と `bool` しかない。さらに、`int` と `bool` は区別されているので、`int` 型の変数に `true` という `bool` 型の定数を代入するとエラーになる。

MiniC は、このように制限されているが、表現力はそれほど劣っているものではなく、たとえば、再帰呼び出しをつかって階乗関数を記述できる。

$ident ::= (\text{英文字からはじまる英数字等の列})$ [識別子]
 $const ::= \text{true} \mid \text{false} \mid (\text{整数の定数})$ [定数]
 $prog ::= dec \mid dec\ prog$ [プログラム]
 $dec ::= \text{type } ident(plist)\ statement$ [関数宣言]
 $\quad \mid \text{type } ident;$ [変数宣言]
 $declist ::= dec \mid dec\ declist$ [宣言の並び]
 $type ::= \text{int} \mid \text{bool}$ [型]
 $plist ::= \text{type } ident \mid \text{type } ident, plist$ [仮引数の並び]
 $statement ::= ident := exp;$ [代入文]
 $\quad \mid \text{while } exp\ statement$ [while 文]
 $\quad \mid \{declist\ stlist\}$ [宣言つき複合文]
 $\quad \mid ;$ [空文]
 $\quad \mid \text{print } exp;$ [print 文]
 $\quad \mid \text{return } exp;$ [return 文]
 $\quad \mid \text{if } exp\ statement\ \text{else } statement$ [if 文]
 $stlist ::= statement \mid statement\ stlist$ [文の並び]
 $exp ::= ident \mid const \mid exp+exp \mid exp*exp \mid$ [式]
 $\quad \mid exp==exp \mid exp>exp$
 $\quad \mid ident(explist) \mid (exp)$
 $explist ::= exp \mid exp, explist$ [式の並び]

図 1: 拡張後の MiniC の構文

```

int f (int x) {
    int result;
    if (x==0) result = 1;
    else result = x * f(x+(-1));
    return result;
}
int main () {
    print f(5);
}

```

ここで print 文のあとには, int 型か bool 型の式を 1 つだけ書けるようにした. また, 全ての関数は, 値を返さなければならない. (void 型がないため.) したがって, 以下のプログラムはエラーである.

```

f (int x) {
    ...

```

```

}
int main () {
    f(5);
}

```

また、関数の返り値を示す `return` 文は、関数の最後になければいけない。(構文上はどこにあらわれてもよいが、関数の最後以外で `return` を実行したときの挙動は定めていない。)

プログラムの構文解析においては、演算子等の優先度が C 言語と同様に定まっている。たとえば、 $1 + 2 * 3$ は $1 + (2 * 3)$ と構文解析される。(上記の構文はをラフに書いたものである。演算子の優先度を考慮にいれて、より精密に記述する方法は、前回学習した。)

3 MiniC 言語の意味論

MiniC は、(構文上の細かな修正は別として) 本質的には以下の点で拡張されているので、それに応じて意味論も拡張する必要がある。

- 変数の宣言がある。(int x; と宣言してから x を使う。)
- 関数の宣言 (定義) があり、関数呼び出しができる。

これらを順番に見ていこう。(ただし、今週は、形式的意味論はやらず、インフォーマルな意味論に留める。)

3.1 大域変数、局所変数、ブロック構造

以下の例題からわかるように、MiniC では、変数の宣言は、4 通りの場所に現れ得る。

```

int x;
int foo (int y) {
    int z;
    ...
    {int w; ...}
    ...
}
int main () {
    ...foo(10) ...
}

```

- 大域変数 (global variable): 上記の x のように、どの関数の中でもない場所で宣言された変数。
- 局所変数 (local variable): 上記の z や w のように、プログラム中の一部の場所からのみで使えるよう宣言された変数。
- 関数の仮引数 (formal parameter): 上記の y のように関数定義に付随して、そのパラメータを表す変数で、関数に局所的な変数である。(参考: $foo(10)$ のように関数 foo の呼び出し側での 10 という引数を、実引数 (actual parameter) という。)

C 言語のように、ALGOL 言語の流れを組む言語 (現在存在する非常に多くの言語) は、プログラム (のテキスト) 全体が、ブロックとよばれる部分に分割される。ブロックについて、基本的な事実を列挙する。

- ブロックは、プログラムの構文上の概念である。(プログラムの構造に対応したものである。)
- ブロックは、入れ子構造をなす。
- 各々の変数宣言は、特定のブロックのみで有効である。(大域変数に対応するブロックは、プログラム全体と見なす。)

これらの事実から、環境をスタックに積んだ形式の処理系 (インタープリタ) が作成できる。(これについては、授業中に MiniC 言語に対する処理系として説明する。)

3.2 変数のスコープ (scope)

変数のスコープとは、その変数の宣言が有効な (プログラム上の) 範囲である。これも構文的に決まる概念である。C 言語の場合、変数宣言は対応するブロックの先頭に置かれるので、スコープはほぼ自明に決まる。

3.3 動的束縛 (dynamic binding)、静的束縛 (static binding)

以下の MiniC プログラムを見てみよう。

```
int x;
int g (int y) {
    return x + y;
}
int f () {
    int x; x = 10;
    return (g(20));
}
int main () {
    x = 5; print f();
}
```

ここで問題となるのは、関数 g における変数 x の使用が、どの宣言に対応するものか、ということである。

もし、字面だけを見ていたら g の中の x は、大域変数である。この値は main 関数で 5 になっているので、x=5 が使われると考えられる。一方、プログラムの実行を追ってみると、関数 g がよばれるのは関数 f からであり、f においては局所的に変数 x が宣言されているので、関数 g の中での x の値は 10 になる。

これらは、両方ともあり得る考え方であり、以下の 2 つの方式に対応する。

- 静的束縛: 変数の使用と、変数の宣言の対応関係 (束縛) は、プログラムを書いた途端に (そのプログラムテキスト上の位置関係により) 決まる。実行時情報にはよらない。
- 動的束縛: 変数の使用と、変数の宣言の対応関係 (束縛) は、プログラムを書いた時ではなく、プログラムの実行時に決まる。

これらの違いについて、その意味、実装、応用の違いを考えてみると面白い。