4N-5

Java クラスファイルを生成する littleC コンパイラの開発

塚本智博　　岩澤京子
拓殖大学工学部情報工学科

1　はじめに

C 言語の学習において，実行時例外のデバッグに多くの時間がとられてしまうことがある．一方，Java 言語の学習においては，実行環境である Java 仮想マシン（以下，JavaVM）により，実行時例外の発生原因とソースプログラム上の位置が表示されるので，デバッグが容易である．

本研究では, C プログラムを JavaVM で実行することにより，JavaVM の機構を利用して有効なデバッグ情報の表示を可能にすることを考えた. そこで, C 言語ソースプログラムを Java クラスファイルに翻訳するコンパイラの開発を行った.

2　システムの機能と構成

2.1　入力言語の制約

本システムの入力として，C プログラム学習者の学習に必要と考えられる機能を持つ littleC 言語を定義した．次に littleC 言語仕様の概要を示す.

① **変数**：基本型には，整数型，実数型を持つ．複合型には，配列型，ポインタ型を持つ.
② **演算子，式，文**：C 言語と同等の機能を持つ
③ **関数**：ユーザ定義の仮引数は，固定数のみ.
④ **ライブラリ関数**：C 言語の入出力関数，文字テスト関数，文字列関数，数学関数を用意した.

2.2　出力

本システムの出力は, Java クラスファイルである. また，JavaVM の例外処理を有効利用するために, Java クラスファイルフォーマットに従ってデバッグ情報を組込む.

2.3　全体構成

本コンパイラは，2 パスコンパイラである．構文木の各ノードは，ソース上の行番号を保持する．この行番号は，デバッグ情報としてオブジェクトコードに組込むときに利用する.

3　変数と配列のポインタの実装

littleC 言語の変数と配列のポインタの実装には, JavaVM において動的に変数領域のアドレスを決定し，アクセスすることができないという問題点があった. 本研究では, 次のようにポインタを実装した.

3.1　ポインタのオブジェクト設計

JavaVM には，C 実行環境におけるポインタ同様のオブジェクト参照がある. これを利用して, littleC 言語のポインタを JavaVM における配列オブジェクトへの参照とそのインデックスで扱うことにした.

JavaVM は，配列アクセス時にレンジオーバーのチェックをおこなう．変数領域に配列オブジェクトを利用することにより, この機能を littleC 言語のポインタアクセス時の範囲チェックに利用可能となった．また，インデックスに対する演算でポインタ演算が可能になった.

3.2　変数領域

littleC 言語のポインタを配列オブジェクトへの参照とそのインデックスで扱うため，ポインタ変数の実装に，java.lang.Object 型と int 型のフィールドを持つ Pointer クラスを導入した.

littleC 言語の全ての変数領域は，表 1のようにデータ型により変数を４つに分類し，配列オブジェクトを対応させて動的に確保することになる. これは, C 言語では&演算子を用いることにより変数のアドレス参照ができるためである．変数領域の構成を図 1に示す.

表 1 データ型と JavaVM 配列オブジェクトの対応

	littleC 言語データ型	JavaVM 配列オブジェクト
①	基本型変数	プリミティブ型一次元配列
②	基本型配列変数	プリミティブ型多次元配列
③	ポインタ型変数	Pointer クラス型一次元配列
④	ポインタ型配列変数	Pointer クラス型多次元配列

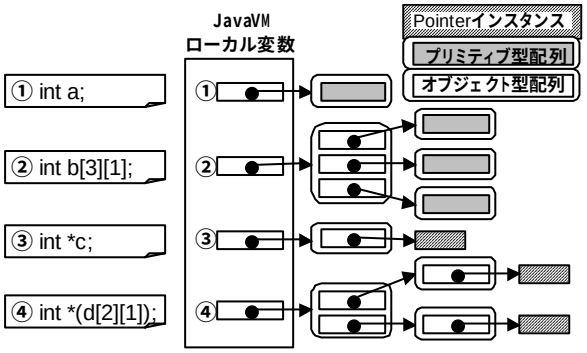

図 1 変数領域の構成

図 1の左側の宣言文①から④は，表 1の対応する

番号①から④の規則により，ヒープに配列で領域を確保することになる．その構成は，右側のようになる．そして，それぞれの配列オブジェクトへの参照を静的に決定したローカル変数①から④に格納する．

4　ライブラリ関数の実装

littleC 言語のライブラリ関数の実装における問題は，可変引数の実装方法である．バイトコードにおいて call 命令の引数は，メソッド・ディスクリプタである．このメソッド・ディスクリプタの表現に可変引数を表すものはないので，JavaVM において可変引数を扱うことはできない．本研究では，次のように可変引数を実装した．

4.1　可変引数のオブジェクト設計と実装

可変引数のオブジェクト設計は，Java5.0[2]で導入された可変引数の実装を参考におこなった．littleC 言語の可変引数の実装に，java.lang.Object 型配列オブジェクトを用いることにした．つまり，動的に実引数の個数分の要素を持つ配列オブジェクトを生成することになる．各要素には，対応する実引数を格納することになる．呼出し時にその参照を渡すことにより，可変引数の実装が可能になった．

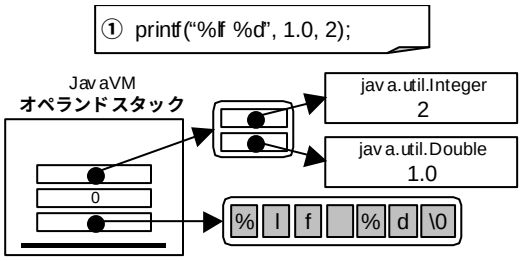

図 2 関数呼出し直前の JavaVM の構造

①の printf 関数の呼出文の場合，関数呼出し直前の JavaVM の構造は，その下のようになる．フォーマット指定である文字列へのポインタをスタックに積み，可変引数を JavaAPI[2] にあるラッパークラスを用いて格納した配列への参照をスタックに積む．また，可変引数がポインタ型である場合，ラッパークラスとして Pointer クラスを用いて配列に格納する．

5　例外処理の実装

littleC 言語の例外処理の実装について述べる．JavaVM による実行において，7 つの JavaVM 実行時例外が発生する可能性がある．そこで，JavaVM から投げられた例外を littleC 言語における発生原因に対応させてエラー表示することにした．その対応を表 2に示す．

表 2 JavaVM 例外と littleC 言語における発生原因

JavaVM 例外	littleC 言語における発生原因
ArithmeticException	算術計算で例外的条件が発生した．
ArrayIndexOutOfBoundsException	範囲違反を犯したポインタアクセスをした．
ClassCastException	可変引数に不正な値を渡した．
NegativeArraySizeException	負のサイズを持った配列をアプリケーションが作成しようとした．
NullPointerException	何も指していないポインタ変数からポインタアクセスをした．
OutOfMemoryError	メモリ不足．
StackOverflowError	再帰の回数が多すぎてスタックオーバーフローが発生した．

実行時例外発生時のエラー表示例を図 3に示す．sample1.c の main 関数の 6 行目から sample2.c の func 関数が呼ばれ，その 5 行目で，配列のレンジオーバーが発生したプログラムの実行例である．

sample1.c

```
1:#include <stdio.h>
2:void main(){
3: int a[5], i;
4: for(i=0;i<5;i++)
5:    a[i] = i;
6: func(a);
7:}
```

sample2.c

```
1:#include <stdio.h>
2:void func(int p[]) {
3: int i;
4: for(i=0;i<=5;i++)
5:    printf("%d ",p[i]);
6:}
```

実行結果

```
0 1 2 3 4
範囲違反を犯したポインタアクセスが発生しました. : 5
        at sample2.func(sample2.c:5)
        at sample1.main(sample1.c:6)
```

図 3 実行時例外発生時のエラー表示例

6　終わりに

littleC 言語ソースプログラムから Java クラスファイルに翻訳する本コンパイラの開発をおこなった．本コンパイラを用いて，バグの原因を特定ができなかった C 言語ソースプログラを Java クラスファイルに翻訳し，JavaVM で実行することで，そのバグが配列のレンジオーバーであったことを特定できた．

本コンパイラは，構造体を導入することで，より多くＣプログラム学習者にとって有効なデバッグツールとして利用可能になると考えられる．

参考文献

[1] 今城哲二, 他：コンパイラとバーチャルマシン，オーム社（2004）

[2] Sun Microsystems： http://java.sun.com/

[3] Jon Meyer, et al.：JAVA Virtual Machine，オライリー・ジャパン（1997）