A note on the implementation of `javar` (version 1.3BETA)*

Aart J.C. Bik and Dennis B. Gannon
Computer Science Department, Indiana University
Lindley Hall 215, Bloomington, Indiana 47405-4101, USA
ajcbik@extreme.indiana.edu

Abstract

This paper briefly describes some implementation details of `javar` which is a prototype Java restructuring compiler that can automatically transform a Java program in which parallel loops or multi-way recursive methods are identified by means of annotations into a form that exploit this parallelism by means of multi-threading.

1 Introduction

The prototype Java restructuring compiler `javar` [1, 2] has been developed at the Indiana University by Aart Bik and Dennis Gannon. The source code of the compiler consists of the following modules:

Makefile	Allows `javar` to be constructed using '`make`' and to be cleaned-up using '`make clean`'
global.h	C-definitions of all global data structures, and function prototypes of all public functions
ast.c	Functions that manipulate the Abstract Syntax Tree
driver.c	Control program for `javar`-output, memory allocation, and command-line arguments processing
html.c	Functions for unparsing the internal program as HTML output
parallel.c	Functions that convert annotated parallel constructs into constructs that use multi-threading
semantic.c	Functions that perform some elementary semantic analysis
symbol.c	Functions that manipulate the symbol table
unparse.c	Functions for unparsing the internal program
parser.y	BISON definitions of the Java parser
scanner.l	FLEX definitions of the Java scanner
lex.yy.c y.tab.c y.tab.h	Files that are automatically generated by FLEX and BISON

In addition, the `javar` package consists of the following directories and files:

*This project is supported by DARPA under contract ARPA F19628-94-C-0057 through a subcontract from Syracuse University.

README LICENSE	Brief summary of `javar` Licensing information
DOC/ANNOUNCE DOC/VERSION DOC/javar.1 DOC/javar_doc.ps DOC/javar_man.ps	Announcement used on newsgroups Summary of modifications with respect to earlier releases of `javar` `man` version of `javar`-manual Complete documentation of the techniques used by `javar` and the results of some preliminary experiments `javar`-manual
EXAMPLES/*.java EXAMPLES/parlooppack/*	Some Java-program examples The `parlooppack` Java-package required by the transformed program

2 Abstract Syntax Tree

The AST (Abstract Syntax Tree) representation of a Java program in `javar` consists of type nodes (`struct type_node`), expression nodes (`struct expr_node`), and statement nodes (`struct stmt_node`). Each newly allocated node in `javar` is linked in a list using the field `next` to simplify garbage collection after a program has been processed. Pointers to type, expression, and statement nodes have the type `type_ptr`, `expr_ptr`, and `stmt_ptr`, respectively.

2.1 Type Nodes

As indicated by the field `kind`, which can have one of the values defined by the enumeration type `types`, each type node either represents `void` (although formally this is not a type in the language specifications [3]), a primitive type, or a reference type (class, interface, and array types). For reference types, field `name` or `aref` of the union `u` is used to represent the basis type of the class or interface type, or array type, respectively.

Unknown types, or the null-type or represented by a pointer with the value `NULL`.

2.2 Expression Nodes

Each expression node has a field `kind` that indicates the kind of expression that is represented by the node. Fields `type` and `lineno` are used to store the type and line number in the program text for this expression, respectively. Moreover, depending on the value of `kind`, one of the fields in the union `u` is used to store further information. Token symbols from the parser are used as possible values for this field:

- Literals (`LITERAL_`): Field `u.lit.s` stores the string representation (to avoid loss in accuracy), while `u.lit.val.l` or `u.lit.val.d` also represent a numerical representation in case `u.lit.set` is set.
- Names (`ID_`): For a qualified name '`nm.id`', fields `u.name.id` and `u.name.entry` store the symbol table entry for the simple name '`id`' and the whole name, respectively. Field `u.name.lcomp` points to the expression node representing '`nm`'. Finally, `u.name.info` and `u.name.scope` represent the function and scope of the name.
- Declaration Statement Expressions ('`d`'): Field `u.declstmt` stores a pointer to the corresponding declaration statement.
- Type Expression Operators (`NEW_`, `INSTOF_`, '`c`' [cast], '`t`' [type-list]): Fields `u.te.tp` and `u.te.ex` are used to link a type and expression node together.

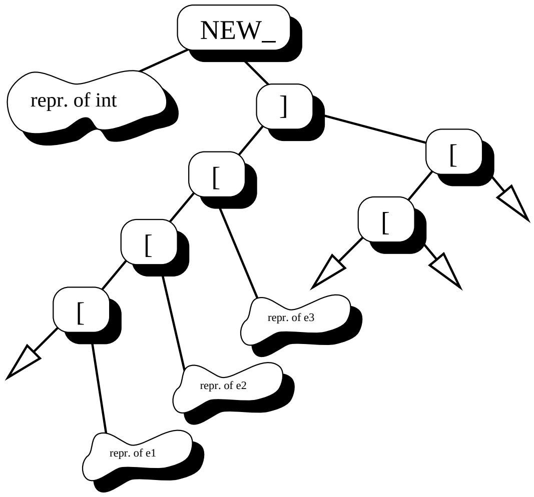

Figure 1: Internal Representation of '`new int[e1][e2][e3][][]`'.

- List Operators: ('`i`' [method-invocation], '{' [array-initializer], '`[`' '`]`' [array-subscripts]) and Ordinary Operators: (EQ_, '+', '-', ...): Fields `u.op.lhs` and `u.op.rhs` are used to represent two other expression nodes.

An array creation expression yields probably the most complex internal representation in `javar`. An example is illustrated in figure 1. Examination of the file `parser.y` may occasionally help in determining how internal representation of particular expressions can be constructed in `javar`.

2.3 Statement Nodes

Each statement node contains integer fields `comment` and `lineno` to represent an entry into a comment buffer (cf. `comment_buffer` in `ast.c`) and a line number in the original Java program text, respectively. In addition, field `kind`, which can have one of the values defined by the enumeration type `stmts`, determines what kind of statement is represented by the node. The following classes of statements can be distinguished:

- Special AST Construct (annotations, concatenations, and labels): Fields in the structure `u.annot`, `u.concat`, or `u.lab` or used to store additional information.
- Simple Statements (`import`, `package`, `break`, ...): Field `arg` is used to represent the single expression argument of such statements.
- Control Statements (`if`, `for`, ...): Fields `u.control.e` and `u.control.scope` represent an expression argument and a scope, respectively. Fields `u.control.s1` and `u.control.s2` can be used to store one or two statement lists that are under control of the control statement.
- Declaration Statements (local and field declarations): Fields `u.decl.tp`, `u.decl.name`, and `u.decl.init` can be used to store the type, name, and, possibly an initializing expression for the declaration. Integer field `u.decl.mod` can be used to represent the modifiers as bit pattern (cf. enumeration type `mods`).
- Headers (blocks, classes, interfaces, constructors, initializers, methods, `catch`-clauses): Fields in the structure `u.header` are used to represent the different constructs required by each header.

The special AST construct for annotations and the headers are probably the most difficult statement representations in `javar` because the fields of the structures `u.annot` and `u.header` are used for different purposes.

For classes, interfaces, methods, and constructors, field `u.header.mod` represents the corresponding modifiers and `u.header.name` the corresponding name. For classes and fields, `u.header.tp` and `u.header.e1` represents the `extends`-type and `implements`-list. For interfaces, `u.header.e1` represents the `extends`-type, whereas for catch-clauses, this field is used to store the argument list. For methods and constructors, fields `u.header.tp`, `u.header.e1`, and `u.header.e2` represent the return type, the formal argument declarations, and the `throws`-list, respectively. In all cases, `u.header.body` represents the associated list of statements, and `u.header.scope` the scope of the construct.

The construct for annotations is used to represent a `par_inv`, `post/wait` or `par` annotation. Because the fields of `u.annot` are used differently for each of these annotations, we recommend to not use this construct for future annotations, but to add a new statement kind in the enumeration type `stmts` and to define a new structure member of the union `u` instead.

3 Symbol Table

In the symbol table, all symbols that occur in a Java program are stored. The organization of the symbol table is illustrated in figure 2. Each symbol is 'hashed' into the data structure `table` using the hash function defined by `hash()` (both local to module `symbol.c`). Symbols that are 'hashed' to the same table entry are chained in a linked list of symbol nodes (`struct symbol_node`), in which the actual symbol is stored in field `symbol`. Field `info` points to a linked list of info nodes (`struct info_node`) that represent the different meanings of the symbol in the program. Each info node has the fields `declstmt`, `scope`, and `info` to store the declaration statement, scope, and determined info of the symbol.

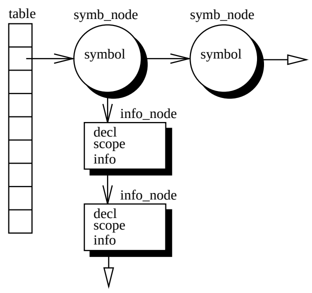

Figure 2: Organization of the Symbol Table

The info field can have one of the value determined by the enumeration type `symbols`:

- `AMBI_N`: The meaning of the symbol has not been determined.[1]
- `PACK_N`: The symbol is used as a package name.
- `TYPE_N`: The symbol is used as a class/interface name.
- `EXPR_N`: The symbol denotes an expression.
- `FIELD_N`: The symbol denotes a field identifier.
- `LOCPAR_N`: The symbol is used as a local variable or parameter.

[1] This meaning can be assigned to a symbol because the exact meaning will be determined at a later stage, or because the exact meaning cannot be determined by the current version of `javar` (which does not account for inheritance and only collects symbol information belonging to a single program file).

- METH_N: The symbol denotes a method identifier.
- LABEL_N: The symbol denotes a label identifier.
- MISCF_N: The symbol denotes an identifier in a field access.
- MISCM_N: The symbol denotes an identifier in a method invocation.

The meaning of names is determined using the methods described in [3, ch6]. Note that in `javar`, a meaning is also assigned to symbols that are not classified as *names* in the Java programming language specifications [3].

4 Concluding Remarks

This prototype version of `javar` supports the Java programming language version 1.0.2 (features of version 1.1. have not been incorporated). The prototype does *not* provide a full front-end for the Java programming language in the sense that unicode escapes are only partly supported and only a limited semantic analysis of the program has been implemented.

New releases of our prototype Java restructuring compiler `javar` (and a prototype bytecode parallelization tool `javab`) will be made available at the HP-Java page at the Indiana University (`http://www.extreme.indiana.edu/hpjava/`). Please send all your comments, bug reports, experiences, and suggestions to: `ajcbik@extreme.indiana.edu`

References

[1] Aart J.C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, 1997.

[2] Aart J.C. Bik and Dennis B. Gannon. `javar` – a prototype java restructuring compiler. Technical Report 487, Computer Science Department, Indiana University, 1997. This manual and the complete source of `javar` are made available at `http://www.extreme.indiana.edu/hpjava/`.

[3] James Gosling, Bill Joy, and Guy Steele. *Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.