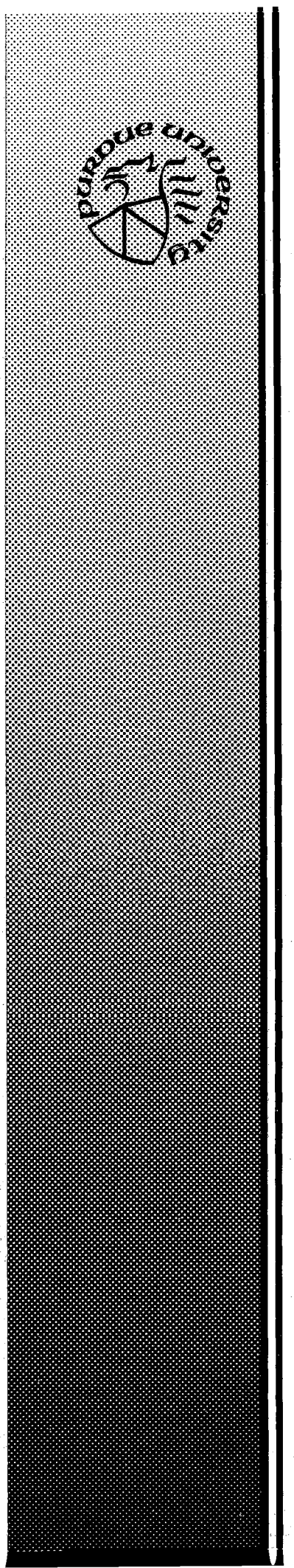
5-1-1990

A Simple Vector Language and its Portable Implementation

Anar Jhaveri
Purdue University

Hank Dietz
Purdue University



A Simple Vector Language and its Portable Implementation

Anar Jhaveri
Hank Dietz

TR-EE 90-41
May 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

A Simple Vector Language and Its Portable Implementation

Anar Jhaveri and Hank Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
jhaveria@ecn.purdue.edu
May 1990

ABSTRACT

Many explicitly parallel languages have been proposed and implemented, but most such languages are complex and are targeted to specific parallel machines. The goal of this project was to design a *very simple*, explicitly parallel, programming language which could easily be implemented and ported to a wide variety of machines. The result was AJL, a structured language with deterministic vector-oriented parallelism.

AJL programs are first compiled into assembly language instructions for an idealized parallel machine, then these assembly language instructions are macro expanded into C code which implements them for the actual target machine. Finally, the target machine's "native" C compiler is used to generate executable code. Macro definitions for "generic" sequential machines have been implemented; macros for the PASM (PArtitionable Simd Mimd) prototype parallel computer are under development.

1. Introduction

AJL was developed to satisfy the need for a very simple, explicitly parallel, easily portable programming language. Most of the parallel languages that exist today, are complex. Porting these languages onto different machines is more difficult not only because of the size of the languages, but also because most parallel languages have been designed to be compatible with only a particular class of parallel machines.

AJL is a simple, structured language. It permits operations on both scalar and vector values and variables. It has a number of special vector-oriented features such as the shuffle, inverse shuffle, right, left, and the ternary operation on vectors. However, all the "vector" operations of AJL are reasonably efficiently executable with or without shared memory under either the MIMD or SIMD execution model. It can be ported with ease onto any parallel or sequential computer; porting AJL onto another machine is just a matter of redefining the C macros that implement the AJL Target machine instructions.

Toward making AJL still more portable and flexible, the AJL compiler was constructed using an EBNF-based LL(1) compiler-compiler system called PCCTS (the Purdue Compiler-Construction Tool Set). As one of the first compilers built using PCCTS, AJL also helped in debugging that system. PCCTS, which was developed by T. Parr, W. Cohen, and H. Dietz, generates both the lexical analyzer and parser from a single, unified, description. Because of this, AJL is very easy to modify or extend.

Actions in the form of C code embedded in the AJL grammar generate instructions for a hypothetical ideal target machine. These instructions are macro expanded by CPP (the C preprocessor) into C code which implements them for the actual target machine. The "native" C compiler for the target machine is used to generate executable code.

Chapter 1 discusses the syntax and semantics of AJL and presents a few sample AJL programs. The AJL target machine language is introduced in Chapter 2. Chapter 3 presents the implementation of AJL. The conclusions are unveiled in Chapter 4. The appendices contain the C language source code for AJL.

1.1. The Language AJL

AJL[1] is a structured, language similar in some respects to C and Pascal. However, unlike C and Pascal, AJL supports vector operations that can be executed in parallel. It also supports operations on scalar variables. It permits the use of global as well as local variables. Functions can be defined within other functions. Vector or `poly` variables can be converted into scalar or `mono` variables and vice versa.

If and while statements are available for loop constructs. The input statement can be used to read input directly from the stdin. Some of the special features include the vector ternary operator, the `left` and `right` shift operators that move all elements of a vector `left` or `right` with wrap around, the `shuf` and `ishuf` operations to shuffle or inverse shuffle the elements of a vector. The size of the no. of elements of a vector can be altered by redefining the value of POLYMAX.

1.2. AJL Syntax

The AJL compiler translates programs written in AJL language into AJL macros, which are the assembly language instructions for an idealized parallel stack machine. The programs written in AJL should adhere to the language description. Following is the syntax specification of the input language, AJL.

[1] AJL stands for Anar Jhaveri's Langauge. This name is a play on the much more serious explicitly-parallel language HJL, also being developed at Purdue.

```
prog : ( decl )* Eof ;
decl : mode WORD ( func | vars ) ;
vars : ( "," WORD )* ";" ;
mode : "mono"
     | "poly" ;
func : " { args } ")" body ;
args : mode WORD ( "," mode WORD )* ;
body : "" ( decl )* ( stat )* "" ;
stat : "" ( stat )* ""
     | "if" x stat { "else" stat }
     | "while" x stat
     | "return" x ";"
     | "print" x ";"
     | WORD assign ";"
     | ";" ;
assign : "=" x
       | "[" x "]" "=" x ;
x : x1 { "?" x1 ":" x1 } ;
x1 : x2 ( x1a )* ;
x1a : "<" x2
    | "<=" x2
    | "<>" x2
    | ">=" x2
    | ">" x2
    | "=" x2 ;
x2 : x3 ( ( "+" | "-" ) x3 )* ;
x3 : x4 ( ( " | "/" ) x4 )* ;
x4 : x5 { "^" x5 } ;
x5 : x6 { ".." x6 } ;
x6 : x7 { "[" x "]" } ;
x7 : "sin" x7
   | "cos" x7
   | "tan" x7
   | "floor" x7
   | "ceil" x7
   | "-" x7
   | " x ")"
   | "left" x7
   | "right" x7
   | "shuf" x7
   | "ishuf" x7
   | mode x7
   | "[" x ( "," x )* "]"
   | WORD ( ( " x ( "," x )* ")" ) | )
   | CONST
   | "pi"
   | "e"
   | "input"
   | "#" ;
```

This language permits operations on scalar or `mono` variables and vector or `poly` variables. Actions are embedded in AJL grammar to generate the equivalent target machine code for

the desired target machine. Actions are blocks of C code enclosed in << and >>. AJL grammar is parsed by ANTLR. ANTLR is a software tool that analyzes such grammar descriptions and produces efficient C programs to recognize phrases in the specified language. This file containing the C program is then compiled using the native C compiler of the machine to give the executable file `ajl`.

1.3. AJL Semantics

The table below briefly summarizes the operations permitted by AJL and the meaning of each of those operations. In the table, the `mono` variables `a` and `b`, and the `poly` variables `c`, `d`, and `e`, are used to illustrate the operand types for each operation. For example, `a + b` represents the addition of two `mono` values, whereas `c + d` represents the addition of two `poly` values.

AJL Construct	Meaning
`mono a,b;`	declaring `mono` (scalar) variables
`poly c,d,e;`	declaring `poly` (vector) variables
`a + b`	add
`a - b`	subtract
`a * b`	multiply
`a / b`	divide
`a ^ b`	power
`sin a`	sine
`cos a`	cosine
`tan a`	tangent
`floor a`	floor
`ceil a`	ceiling
`a < b`	less than comparison; 1 if true, 0 otherwise
`a .. b`	linear range from `a` to `b` as a vector
`poly a`	equivalent to `a .. a`
`c[a]`	`a+1`th element of vector `c`
`c + d`	vector add
`c - d`	vector sub
`c * d`	vector multiply
`c / d`	vector divide
`c ^ d`	vector power
`sin c`	vector sine
`cos c`	vector cosine
`tan c`	vector tangent
`floor c`	vector floor
`ceil c`	vector ceiling
`[a, b, ...]`	vector value list, e.g., `c = [1, 2, 3, 4, 5];`
`c < d`	element-by-element less than comparison
`c ? d: e`	vector ternary operator
`left c`	vector shift by 1, e.g., `c[x] = c[x+1]` with wrap
`right c`	vector shift by 1, e.g., `c[x] = c[x-1]` with wrap
`shuf c`	vector shuffle
`ishuf c`	vector inverse shuffle
`pi`	built-in constant value of π
`e`	built-in constant value of e
`#`	number of elements in a vector (poly)
`input`	obtain a `mono` value from `stdin`

While the above summary suffices to introduce most of the conventional operations, some of the operators are unfamiliar and require additional explanation:

`a .. b`

This creates a vector in which the elements have values ranging from `a` to `b`. The element values are computed by making each element *i* have the value `a+`*i*`*((b-a)/(#-1))`. In other words, the elements are equivalent to values sampled at regular spacing on a line drawn from `a` to `b`. This is particularly useful for constructing vectors representing linear functions for numerical analysis applications.

`[a, b, ...]`

This creates a vector where the element values are `a`, `b`, etc. This is not currently

implemented.

`c? d: e`

This is the vector ternary operator — effectively, an element-by-element conditional substitution. For each element of `c` which is non-zero, the corresponding element of `d` is returned; for the elements of `c` which are zero, the corresponding element of `e` is returned.

`left c`

This shifts each element of the vector c left by one, with wrap around — the inverse of the `right` operation. E.g., if `#==4`, the vector `left [0, 1, 2, 3]` would be `[1, 2, 3, 0]`.

`right c`

This shifts each element of the vector c right by one, with wrap around — the inverse of the `left` operation. E.g., if `#==4`, the vector `right [0, 1, 2, 3]` would be `[3, 0, 1, 2]`.

`shuf c`

This shuffles the elements of the vector `c` using a formula derived from that on page 489, "Ultracomputers", by J.T. Schwartz, ACM Transactions on Programming Languages and Systems, Oct. 1980. This is commonly used in algorithms such as FFT. E.g., if `#==8`, the vector `shuf [0, 1, 2, 3, 4, 5, 6, 7]` would be `[0, 2, 4, 6, 1, 3, 5, 7]`.

`ishuf c`

This inverse shuffles the elements of the vector `c` using a formula derived from that on page 489, "Ultracomputers", by J.T. Schwartz, ACM Transactions on Programming Languages and Systems, Oct. 1980. E.g., if `#==8`, the vector `ishuf [0, 1, 2, 3, 4, 5, 6, 7]` would be `[0, 4, 1, 5, 2, 6, 3, 7]`.

`#` This built-in constant gives the total number of elements in a `poly` variable. Notice that these elements are indexed as `0` to `#-1`. Although `#` cannot be modified during program execution, it can be redefined by the user at compile time. To make `#` be *x* instead of the default value, one would simply include the following line in the C program generated by AJL:

```
#define POLYMAX x
```

`input`

This "variable", when examined, will read a single number from the standard input and will return that value as a `mono`. White space, etc., are ignored as per the rules of C's `scanf` function.

AJL allows the use of local as well as global variables in functions. Nested scoping is allowed. The variable/function name, whether it is local or global, mono or poly, a keyword or not, and its offset relative to its position in the present active frame, comprise the information that is stored in the symbol table.

1.4. Sample AJL Programs

A few sample AJL programs, the resulting target machine code and final output are described below. In order to understand fully how the final output is generated, the user will find it helpful to go over the target machine code and note the operation performed by each instruction.

1.4.1. Mono Factorial

This sample program, `example1.ajl`, calculates the factorial of a scalar input. This program accepts a `mono` variable input from stdin and computes the factorial of that number by recursively calling the fact program. The final result is the factorial of the number accepted as input from the user.

```
mono fact(mono n)
{
    if (n < 2) return(1);
    return(n * fact(n-1));
}

mono main()
{
    print fact(input);
}
```

Resulting target machine code for the above sample program, `example1.c`.

```
#include "ajl_sup.h"

fact()
{
     MFLD(0);
     MCONST(2);
     MLT;
     IF {
     MCONST(1);
     MRET;
     }
     MFLD(0);
     MARK;
     MFLD(0);
     MCONST(1);
     MSUB;
     FRAME;
     fact();
     MMUL;
     MRET;
}

real_main()
{
     MARK;
     INPUT;
     FRAME;
     fact();
     MPRINT;
}


int
main()
{
     fp->f_mono = msp;
     fp->f_poly = psp;
     real_main();
     return((int) msp[-1]);
}
```

The input and resulting output after compiling `example1.c` are given below. Text typed by the user is highlighted by using **`this font`**. Output and prompts from the computer are shown in `this font`. Comments about the output are given in *this font.*

```
$ example1
5
120
$ example1
20
2.4329e+18
$
```

1.4.2. Vector Operations Demonstration

This sample program, `example2.ajl`, performs a left shift, right shift, shuffle and inverse shuffle operations, in that order, on the vector `x`. The resulting vector after this sequence of operations is the same as the original vector `x`. A subtraction operation at this stage, hence, results in a zero vector which is assigned to `x`. The 0th, 1st and 4th elements of the vector `x` are then altered by assigning them individual values. The vector ternary operator then tests each element of the vector `x`. If the element is zero, it prints it, else it prints the corresponding element of the other vector, namely, `one`. Intermediate results after the main operations are printed to enable the reader to understand better the actions performed by these operations.

```
mono main()
{
    poly x;
    poly one;
    one = 1;
    x = 1 .. #;
    print x;
    x = left(x);
    print x;
    x = right(x);
    print x;
    x = shuf(x);
    print x;
    x = ishuf(x);
    print x;
    x = ishuf(shuf(right(left(x)))) - x;
    print x;
    x[0] = 3;
    x[1] = 5;
    x[4] = 2;
    print x;
    print x ? one: x;
}
```

The resulting target machine code for the above sample program is `example2.c`.

```
#include "ajl_sup.h"

real_main()
{
    MCONST(0);
    PCONST;
    MCONST(0);
    PCONST;
    MCONST(1);
    PCONST;
    PFST(1);
    MCONST(1);
    MCONST(POLYMAX);
    PRANGE;
    PFST(0);
    PFLD(0);
    PPRINT;
    PFLD(0);
    PLEFT;
    PFST(0);
    PFLD(0);
    PPRINT;
    PFLD(0);
    PRIGHT;
    PFST(0);
    PFLD(0);
    PPRINT;
    PFLD(0);
    PSHUF;
    PFST(0);
    PFLD(0);
    PPRINT;
    PFLD(0);
    PISHUF;
    PFST(0);
    PFLD(0);
    PPRINT;
    PFLD(0);
    PLEFT;
    PRIGHT;
    PSHUF;
    PISHUF;
    PFLD(0);
    PSUB;
    PFST(0);
    PFLD(0);
    PPRINT;
    MCONST(0);
    MCONST(3);
    PFLD(0);
    PSUBST;
    PFST(0);
    MCONST(1);
```

```
    MCONST(5);
    PFLD(0);
    PSUBST;
    PFST(0);
    MCONST(4);
    MCONST(2);
    PFLD(0);
    PSUBST;
    PFST(0);
    PFLD(0);
    PPRINT;
    PFLD(0);
    PFLD(1);
    PFLD(0);
    POLYIF;
    PPRINT;
}

int
main()
{
    fp->f_mono = msp;
    fp->f_poly = psp;
    real_main();
    return((int) msp[-1]);
}
```

The resulting output after compiling `example2.c` is as follows:

```
$ example2
0:1     4:5     8:9     12:13  value of x
1:2     5:6     9:10    13:14
2:3     6:7     10:11   14:15
3:4     7:8     11:12   15:16


0:2     4:6     8:10    12:14  value of left(x)
1:3     5:7     9:11    13:15
2:4     6:8     10:12   14:16
3:5     7:9     11:13   15:1


0:1     4:5     8:9     12:13  value of right(x)
1:2     5:6     9:10    13:14
2:3     6:7     10:11   14:15
3:4     7:8     11:12   15:16


0:1     4:3     8:5     12:7   value of shuf(x)
1:9     5:11    9:13    13:15
2:2     6:4     10:6    14:8
3:10    7:12    11:14   15:16


0:1     4:5     8:9     12:13  value of ishuf(x)
1:2     5:6     9:10    13:14
2:3     6:7     10:11   14:15
3:4     7:8     11:12   15:16


0:0     4:0     8:0     12:0   value of x minus x
1:0     5:0     9:0     13:0
2:0     6:0     10:0    14:0
3:0     7:0     11:0    15:0


0:3     4:2     8:0     12:0   value of altered x
1:5     5:0     9:0     13:0
2:0     6:0     10:0    14:0
3:0     7:0     11:0    15:0


0:1     4:1     8:0     12:0   value after ? : operation
1:1     5:0     9:0     13:0
2:0     6:0     10:0    14:0
3:0     7:0     11:0    15:0
$
```

1.4.3. Vector Summation

This sample program, `example3.ajl`, operates on both `mono` and `poly` variables. It takes two scalars as input and creates a vector with a value which is a linear range from the first to the second. It then sums up all the elements of the resulting vector using "recursive doubling" (in compiler parlance, a tree-structured associative reduction).

```
poly global_thingy;

poly
left_by_n(poly one, mono two)
{
    if (two > 0) {
        return( left_by_n(left(one), (two)-1) );
    } else {
        return(one);
    }
}

mono
sum(poly one)
{
    mono i;
    poly t;

    t = one;
    i = 1;
    while (i < #) {
        t = t + left_by_n(t, i);
        i = 2 * i;
    }
    return( t[ 0 ] );
}

mono
main()
{
    global_thingy = input..input;
    print global_thingy;
    print sum(global_thingy);
}
```

The resulting target machine code for the above sample program is `example3.c`.

```
#include "ajl_sup.h"

left_by_n()
{
     MFLD(0);
     MCONST(0);
     MGT;
     IF {
     MARK;
     PFLD(0);
     PLEFT;
     MFLD(0);
     MCONST(1);
     MSUB;
     FRAME;
     left_by_n();
     PRET;
     } else {
     PFLD(0);
     PRET;
     }
}

sum()
{
     MCONST(0);
     MCONST(0);
     PCONST;
     PFLD(0);
     PFST(1);
     MCONST(1);
     MFST(0);
     WHILE
     MFLD(0);
     MCONST(POLYMAX);
     MLT;
     DO
     PFLD(1);
     MARK;
     PFLD(1);
     MFLD(0);
     FRAME;
     left_by_n();
     PADD;
     PFST(1);
     MCONST(2);
     MFLD(0);
     MMUL;
     MFST(0);
     }
     PFLD(1);
     MCONST(0);
     PSUBLD;
```

```
    MRET;
}

real_main()
{
    INPUT;
    INPUT;
    PRANGE;
    PST(0);
    PLD(0);
    PPRINT;
    MARK;
    PLD(0);
    FRAME;
    sum();
    MPRINT;
}


int
main()
{
    MCONST(0); PCONST;
    fp->f_mono = msp;
    fp->f_poly = psp;
    real_main();
    return((int) msp[-1]);
}
```

Running this program yields:

```
$ example3
1 16

0:1     4:5     8:9     12:13   value of global_thingy
1:2     5:6     9:10    13:14
2:3     6:7     10:11   14:15
3:4     7:8     11:12   15:16

136                             value of sum
$
```

2. AJL Target Language

AJL is a simple vector calculator language designed to be easily implemented and used on PASM and other parallel or serial machines. The language compiles into C code with various macro references, and porting the language is simply a matter of changing the macro definitions to reflect the target architecture.

A set of macros for executing AJL code within a single process under any flavor of UNIX system has been developed. Macro definitions for PASM are underway.

This document simply outlines the function of each macro, hence serving as an implementor's guide in porting AJL to other target machines.

2.1. Terminology

In the interest of brevity, a few terms are used within following descriptions. These terms are:

`PSP` Poly stack pointer

`MSP` Mono stack pointer

`POLYMAX` No. of elements in a poly

Second on the stack
This refers to the position that is below the topmost element on the stack.

mx Mono expression

mx1 Mono expression in the second place on the `mono` stack

mx2 Mono expression in the topmost place on the `mono` stack

2.2. Poly Macros

The following macros implement the poly (vector) operations.

`forpoly(v)`
This is a simple `for` loop. The variable *v* takes on integer values from 0 to `POLYMAX`-1 in increments of one.

`PPRINT` Pretty prints the vector topmost on the poly stack on an element-by-element basis. The vector is then removed from the poly stack.

`POLYIF` This is the vector ternary operator. Each element of the vector which is third on the stack is tested. If it is true, then the resulting element is assigned the value of the element of the vector second on the stack else it is assigned the value of the element of the vector topmost on the poly stack. The top 3 vectors are then removed from the stack and replaced by the result.

`PLT` Compares the top two poly stack vectors element-by-element for less than.These two polys are removed from the poly stack and replaced by a single poly result which has a value of 1 for each element where the first poly element was less than the second, 0

otherwise.

`PLE` Compares the top two poly stack vectors element-by-element for less than or equal to. These two polys are removed from the poly stack and replaced by a single poly result which has a value of 1 for each element where the first poly element was less than or equal to the second, 0 otherwise.

`PNE` Compares the top two poly stack vectors element-by-element for not equal to. These two polys are removed from the poly stack and replaced by a single poly result which has a value of 1 for each element where the first poly element was not equal to the second, 0 otherwise.

`PGE` Compares the top two poly stack vectors element-by-element for greater than or equal to. These two polys are removed from the poly stack and replaced by a single poly result which has a value of 1 for each element where the first poly element was greater than or equal to the second, 0 otherwise.

`PGT` Compares the top two poly stack vectors element-by-element for greater than. These two polys are removed from the poly stack and replaced by a single poly result which has a value of 1 for each element where the first poly element was greater than the second, 0 otherwise.

`PEQ` Compares the top two poly stack vectors element-by-element for equality. These two polys are removed from the poly stack and replaced by a single poly result which has a value of 1 for each element where the first poly element was equal to the second, 0 otherwise.

`PADD` Adds the top 2 poly stack vectors element-by-element. These 2 polys are removed from the stack and replaced by the resulting vector.

`PSUB` Subtracts the top 2 poly stack vectors element-by-element. These 2 polys are removed from the stack and replaced by the resulting vector.

`PMUL` Multiplies the top 2 poly stack vectors element-by-element. These 2 polys are removed from the stack and replaced by the resulting vector.

`PDIV` Divides the top 2 poly stack vectors element-by-element. These 2 polys are removed from the stack and replaced by the resulting vector.

`PPOW` Raises the second vector on the stack to the power of the first vector. The top 2 vectors are removed from the poly stack and replaced by the result.

`PSIN` Performs the sine operation on an element-by-element basis on the vector topmost on the poly stack. This vector is then replaced by the result.

`PCOS` Performs the cosine operation on an element-by-element basis on the vector topmost on the poly stack. This vector is then replaced by the result.

`PTAN` Performs the tangent operation on an element-by-element basis on the vector topmost on the poly stack. This vector is then replaced by the result.

`PNEG` Negates each element of the vector topmost on the poly stack.

`PFLOOR` Computes the vector floor. The floor operation is performed element-by-element on the vector topmost on the poly stack giving the result vector.

`PCEIL` Computes the vector ceiling. The ceiling operation is performed element-by-element on the vector topmost on the poly stack giving the result vector.

`PLEFT` This is a vector shift left operation. All the elements of the vector topmost on the stack are shifted left by one place with wrap around.

`PRIGHT` This is a vector right shift operation. All the elements of the vector topmost on the stack are shifted right by one place with wrap around.

`PSHUF` This shuffles the elements of the vector using the formula derived from that on page 489, "Ultracomputers", by J.T. Schwartz, ACM Transactions on Programming Languages and Systems, Oct. 1980.

`PISHUF` This does the inverse shuffle of the elements of the vector using the formula derived from that on page 489, "Ultracomputers", by J.T. Schwartz, ACM Transactions on Programming Languages and Systems, Oct. 1980.

`PRANGE` This forms a linear range from *mx1* to *mx2* as a vector. It takes the values of the 2 monoexpressions from the mono stack and generates a vector on the poly stack whose elements have values starting from the value of *mx1* up to *mx2* in equal increments. The 2 monoexpressions are then removed from the mono stack.

`PCONST` This forms a linear range from *mx2* to *mx2* as a vector. It takes the value of the monoexpression *mx2* from the top of the stack and generates a vector on the poly stack each of whose elements have the value of *mx2*. The mono stack element is then removed.

`PSUBST` This assigns a value to a single element of a vector on top of the poly stack. The value of *mx1*, which is the monoexpression below the one on top of the stack is used to select an element offset by that value in the vector. The value of *mx2*, which is on top of the stack, is then assigned to the chosen element of the vector. The 2 mono stack elements are then popped off the stack.

`PSUBLD` This is used to put a mono value on top of the mono stack from the poly stack. The *mx* on top of the mono stack is used to select an element offset by that value in the vector on the poly stack and the contents of this element replace the *mxfP on the mono stack.*

`PLD (`*x*`)` This moves a vector to the top of the poly stack from a location offset by *x* from the starting address of the poly stack. This is typically used with global variables.

`PST (`*x*`)` This pops off an element from on top of the poly stack and places it in a location on the poly stack that is offset by *x* from the starting address of the poly stack. This is typically used with global variables.

`PFLD(x)` This moves a vector to the top of the poly stack from a location offset by *x* from the starting address of the frame pointed to by the frame pointer, which is the currently active frame. The contents of this location are pushed onto the top of the poly stack. This is typically used for local variables.

`PFST(x)` Here *x* is used to access the location offset by *x* in the frame pointed at by the frame pointer, which is the currently active frame. The top vector of the poly stack is then stored at this location. This is typically used for local variables.

2.3. Frame Manipulation Macros

`PRET` Performs a poly return. It resets the mono and poly stack pointers. It puts the poly value to be returned on top of the poly stack. It decrements the frame pointer thereby discarding the previous frame.

`MAXFRAMES`

This defines the maximum no. of frames that can be active at any given time within a program. It can be altered by the user by redefining it, if the user so desires.

`MARK`

`FRAME` The above stated 2 macros result in setting the frame pointer to point to a new frame. This new frame is created at the time that a function is called and it stores the arguments passed to the function as well as the variables local to that function at a positive offset from the frame pointer.

2.4. Mono Macros

The following macros implement the mono (scalar) operations.

`IF` Checks whether the topmost element on the mono stack evaluates to non-zero. The element is then removed from the mono stack.

`WHILE`

`DO`

The substatement is executed repeatedly so long as the value of the element topmost on the mono stack remains non-zero. The test takes place before each execution of the statement. The element is then removed from the mono stack.

`POLYMAX` This defines the no. of elements that can be present in a vector. It can be redefined by the user if necessary.

`MPRINT` Prints the element topmost on the mono stack and then removes the element from the stack.

`MLT` Compares the top 2 elements of the mono stack for less than. These 2 elements are removed from the mono stack and replaced by a single mono result which has a value of 1 if the first element was less than the second, which is the element on top of the stack, 0 else.

`MLE` Compares the top 2 elements of the mono stack for less than or equal to. These 2 elements are removed from the mono stack and replaced by a single mono result which has a value of 1 if the first element was less than or equal to the second, which is the element on the top of the stack, 0 else.

`MNE` Compares the top 2 elements of the mono stack for not equal to. These 2 elements are removed from the mono stack and replaced by a single mono result which has a value of 1 if the first element was not equal to the second, the element on top of the stack, 0 else.

`MGE` Compares the top 2 elements of the mono stack for greater than or equal to. These 2 elements are removed from the mono stack and replaced by a single mono result which has a value of 1 if the first element was greater than or equal to the second, the element on top of the stack, 0 else.

`MGT` Compares the top 2 elements of the mono stack for greater than. These 2 elements are removed from the mono stack and replaced by a single mono result which has a value of 1 if the first element was greater than the second, the element on top of the stack, 0 else.

`MEQ` Compares the top 2 elements of the mono stack for equality. These 2 elements are removed from the mono stack and replaced by a single mono result which has a value of 1 if the first element was equal to the second, the element on top of the stack, 0 else.

`MADD` Adds the top 2 mono stack elements. These are then removed and replaced by the result.

`MSUB` Subtracts the top 2 mono stack elements. These are then removed and replaced by the result.

`MMUL` Multiplies the top 2 mono stack elements. These are then removed and replaced by the result.

`MDIV` Divides the top 2 mono stack elements. These are then removed and replaced by the result.

`MNEG` Negates the element on top of the mono stack.

`MPOW` Raises the first element to the power of the second element, which is on top of the mono stack. The top 2 elements are then removed and replaced by the result.

`MSIN` Performs the sine operation on the element topmost on the mono stack. This element is then replaced by the result.

`MCOS` Performs the cosine operation on the element topmost on the mono stack. This element is then replaced by the result.

`MTAN` Performs the tangent operation on the element topmost on the mono stack. This element is then replaced by the result.

`MFLOOR` Computes the floor of the element topmost on the mono stack and replaces the element by the result.

`MCEIL` Computes the ceiling of the element topmost on the mono stack and replaces the element by the result.

`MCONST` (*x*)
Pushes the constant x on the mono stack.

`MLD` (*x*) Pushes the element offset by *x* from the starting address of the mono stack onto the top of the mono stack.

`MST` (*x*) Removes the topmost element of the mono stack and places it in a location on the mono stack that is offset by *x* from the starting address of the stack.

`MFLD` (*x*) Copies the element offset by *x* in the current frame to the top of the mono stack.

`MFST` (*x*) Stores the element on top of the mono stack in a location offset by *x* in the current frame. The same element on top of the stack is then popped off.

`MRET` Performs a mono return. It resets the mono and poly stack pointers. It puts the mono value to be returned on top of the mono stack. It decrements the frame pointer thereby discarding the previous frame.

2.5. Predefined Variables

`INPUT` Reads a mono value from the standard input and pushes it onto the mono stack.

`M_PI` The constant π.

`M_E` The constant *e*.

`MAKEMONO`
This creates a mono variable from a vector by pushing the first element of the vector on top of the poly stack onto the mono stack. The vector on the poly stack is then popped off.

3. AJL Implementation

The single-process unix-based version of AJL is currently maintained on a Sun workstation. On that machine, an executable AJL compiler, `ajl`, would be created according to the following makefile:

```
CFLAGS = -I/home/aquarium3/carp/ANTLR/h -g
ANTLRFLAGS = -I

SRC = ajl.c scan.c sym_table.c err.c
OBJ = ajl.o scan.o sym_table.o err.o

ajl : $(OBJ) $(SRC)
    cc -g -o ajl $(CFLAGS) $(OBJ)
ajl.c : ajl.g
    /home/aquarium3/carp/bin/antlr $(ANTLRFLAGS) ajl.g
scan.c : parser.dlg ajl.g
    /home/aquarium3/carp/bin/dlg -C2 parser.dlg  scan.c
```

AJL programs serve as input to the executable file `ajl` and the output is a C program containing the target machine code for the ideal machine. This C program can then be compiled using the native C compiler of the machine to generate the desired output. There is also an ajl support file, `ajl_sup.c`, which has to be compiled along with the above. For example, if the input file containing the AJL program was called `v0.ajl`, then the following commands generate the desired executable file `v0`:

```
ajl < v0.ajl > v0.c
cc v0.c ajl_sup.c -o v0
```

A brief description of various files associated with AJL is listed below. The sources are listed in appendixes A-E.

Appendix A
: The file `ajl.g`; the AJL syntax specification

Appendix B
: The file `ajl.h`; extern declarations of global variables used in `ajl.g` actions

Appendix C
: The file `ajl_sup.h`; this is an include file containing the types, extern declarations, and macros for AJL programs running under UNIX.

Appendix D
: The file `ajl_sup.c`; the global data intialization and support functions for AJL programs running under UNIX.

Appendix E
: The file `sym_table.c`; the symbol table manager for AJL.

4. Conclusions

AJL has been successfully implemented on the UNIX operating system. Porting it to any parallel computer should be accomplished without much trouble. It is just a matter of redefining the macros used to implement the target language. In most cases, the only changes are in the implementations of the communication instructions, such as `LEFT`.

An initial, untested, set of macro definitions for AJL to run on the PASM (PArtitionable Simd Mimd) prototype parallel computer have already been developed and further work is going on in this area.

Further, as one of the first compilers built using PCCTS, AJL served to help debug the system, and has helped show the utility of PCCTS.

5. Acknowledgements

Will Cohen and Terence Parr, PhD students in Electrical Engineering at Purdue, deserve special thanks for always being there to help me with any problems that I had with PCCTS (Purdue Compiler-Construction Tool Set) — particularly with the pre-β release versions.

Appendix A: `ajl.g`

```
/*  AJL syntax specification

    May 1990, A. Jhaveri & H. Dietz
*/
#attrib <<
    #include <stdio.h>
    #define D_Text
    #include "ajl.h"
    >>

<<
Attrib  MONO    = { "mono" };
Attrib  POLY    = { "poly" };

#define ISMONO(x)   (!strcmp("mono",x.text))
#define ISPOLY(x)   (!strcmp("poly",x.text))

>>

#token  KVAR
#token  KFUNC

#token "[\t\ ]"     << LexSkip(); >>                /* Ignore White */
#token "[\n]"       << lex_line++; LexSkip(); >>    /* Track Line # */

prog:   << /* syminit(); */
        g_ptr = sp = 0; mg_cnt=0; pg_cnt=0;
        frame_ptr_array[frame_ptr++] = sp;
        type = GLOBAL;
        printf("#include \"ajl_sup.h\"\n\n");
    >>
    (decl)* "@"
    <<
        printf("\nint\nmain()\n{\n");
        {
            /* Save space for globals... */
            register int i;

            for (i=0; i<pg_cnt; ++i) {
                printf("\tMCONST(0); PCONST;\n");
            }
            for (i=0; i<mg_cnt; ++i) {
                printf("\tMCONST(0);\n");
            }
        }
        printf("\tfp->f_mono = msp;\n");
        printf("\tfp->f_poly = psp;\n");
        printf("\treal_main();\n");
        printf("\treturn((int) msp[-1]);\n}\n");
    >>
    ;
```

```
decl:   mode    WORD
    <<
        mode_flg = 0;
        strcpy(temp, $2.text);
        if (ISMONO($1)) {
            mode_flg = MONOWD;
            }
        else {
            mode_flg = POLYWD;
        }
    >>
    (func[$0] | vars)[$2]
    ;

vars:   <<
        if (mode_flg == MONOWD) {
            if (type == GLOBAL) {
                enter(temp, KVAR, type, MONOWD,mg_cnt);
                mg_cnt++;
            } else {
                printf("\tMCONST(0);\n");
                enter(temp, KVAR, type, MONOWD,ml_cnt);
                ml_cnt++;
            }
        } else if(mode_flg == POLYWD) {
            if (type == GLOBAL) {
                enter(temp, KVAR, type, POLYWD,pg_cnt);
                pg_cnt++;
            } else {
                printf("\tMCONST(0);\n");
                printf("\tPCONST;\n");
                enter(temp, KVAR, type, POLYWD, pl_cnt);
                pl_cnt++;
            }
        }
    >>
    (","    WORD
    <<
        if (mode_flg == MONOWD) {
            if (type == GLOBAL) {
                enter($2.text, KVAR, type, MONOWD,mg_cnt);
                mg_cnt++;
            } else {
                printf("\tMCONST(0);\n");
                enter($2.text, KVAR, type, MONOWD,ml_cnt);
                ml_cnt++;
            }
        } else if(mode_flg == POLYWD) {
            if (type == GLOBAL) {
                enter($2.text, KVAR, type, POLYWD,pg_cnt);
                pg_cnt++;
            } else {
                printf("\tMCONST(0);\n");
                printf("\tPCONST;\n");
```

```
                enter($2.text, KVAR, type, POLYWD, pl_cnt);
                pl_cnt++;
            }
        }
    >>
    )*  ";"
    ;

mode:   "mono"
    <<
    strcpy($0.text,"mono");
    >>
    |   "poly"
    <<
    strcpy($0.text,"poly");
    >>
    ;

func:   "\("  <<
        type = LOCAL;
        frame_ptr_array[frame_ptr++] = sp;
        l_ptr = sp; ml_cnt=0; pl_cnt=0;
        enter($0.text, KFUNC, type, mode_flg,0);
        if (strcmp($0.text, "main") == 0) {
            printf("real_%s()\n{\n",$0.text);
        } else {
            printf("%s()\n{\n", $0.text);
        }
    >>
    {args}  "\)"    body
    <<
        printf("}\n\n");
        sp =l_ptr+1; frame_ptr--;
        if (frame_ptr > 1) {
            l_ptr = frame_ptr_array[frame_ptr-1];
        }
    >>
    ;

args:   mode    WORD
    <<
        if (ISMONO($1)) {
                mode_flg = MONOWD;
                enter($2.text, KVAR, type, mode_flg,ml_cnt);
                ml_cnt++;
                }
        else {  mode_flg =POLYWD;
            enter($2.text, KVAR, type, mode_flg,pl_cnt);
            pl_cnt++;
        }
    >>
    (","    mode    WORD
    <<
        if (ISMONO($2)) {
```

```
                mode_flg = MONOWD;
                enter($3.text, KVAR, type, mode_flg,ml_cnt);
                ml_cnt++;
                }
        else {
            mode_flg =POLYWD;
            enter($3.text, KVAR, type, mode_flg,pl_cnt);
            pl_cnt++;
        }
    >>
    )*
    ;

body:   "\{"
    <<
        type = LOCAL;
    >>
    (decl)* (stat)* "\}"
    ;

stat:   "\{"    (stat)* "\}"
    |   "if"    x
    <<
        printf("\tIF {\n");
    >>
    stat    {"else"
    <<
        printf("\t} else {\n");
    >>
    stat}
    <<
        printf("\t}\n");
    >>
    |   "while"
    <<
        printf("\tWHILE\n");
    >>
    x
    <<
        printf("\tDO\n");
    >>
    stat
    <<
        printf("\t}\n");
    >>
    |   "return" x ";"
    <<
        if (ISMONO($2)) {
            printf("\tMRET;\n");
        } else {
            printf("\tPRET;\n");
        }
    >>
    |   "print" x ";"
```

```
    <<
        if (ISMONO($2)) {
            printf("\tMPRINT;\n");
        } else {
            printf("\tPPRINT;\n");
        }
    >>
    |   WORD assign[$1] ";"
    |   ";"
    ;

assign: "=" x
    <<
        note = lookup($0.text);

        if ((modes[note] == POLYWD) && ISMONO($2)) {
            /* Promote x to a POLY value */
            printf("\tPCONST;\n");
            $2 = POLY;
        } else if ((modes[note] == MONOWD) && ISPOLY($2)) {
            error("cannot assign a poly value to a mono variable");
        }

        switch (modes[note]) {
        case MONOWD:
            switch (scope[note]) {
            case GLOBAL:
                printf("\tMST(%d);\n", offsets[note]);
                break;
            case LOCAL:
                printf("\tMFST(%d);\n", offsets[note]);
            }
            break;
        case POLYWD:
            switch (scope[note]) {
            case GLOBAL:
                printf("\tPST(%d);\n", offsets[note]);
                break;
            case LOCAL:
                printf("\tPFST(%d);\n", offsets[note]);
            }
            break;
        }
    >>
      | "\["    x   "\]"    "=" x
    <<
        flag = off;
        if (!(ISMONO($5))) error("%s operand should be mono value",$5);
        oldoffset = lookup($0.text);
        if (scope[oldoffset]==GLOBAL) {
            printf("\tPLD(%d);\n",offsets[oldoffset]);
            flag = on;
        }
        else {
```

```
            printf("\tPFLD(%d);\n",offsets[oldoffset]);
        }
        printf("\tPSUBST;\n");
        if (flag == on) {
                flag = off;
                printf("\tPST(%d);\n",offsets[oldoffset]);
                }
        else        printf("\tPFST(%d);\n",offsets[oldoffset]);
    >>
      ;

x:  x1 {"?"
    x1  ":"
    x1
    <<if (!ISMONO($0)) printf("\tPOLYIF;\n"); >>
    }[$1] << $0=$1; >>
 ;

x1: x2 (x1a)*[$1] << $0=$1; >>
  ;

x1a:    "<" x2
    <<
        type_match("<", $0, $2);
        printf("\t%cLT;\n", (ISMONO($2) ? 'M' : 'P'));
    >>
   |    "<=" x2
    <<
        type_match("<=", $0, $2);
        printf("\t%cLE;\n", (ISMONO($2) ? 'M' : 'P'));
    >>
   |    "<>" x2
    <<
        type_match("<>", $0, $2);
        printf("\t%cNE;\n", (ISMONO($2) ? 'M' : 'P'));
    >>
   |    ">=" x2
    <<
        type_match(">=", $0, $2);
        printf("\t%cGE;\n", (ISMONO($2) ? 'M' : 'P'));
    >>
   |    ">" x2
    <<
        type_match(">", $0, $2);
        printf("\t%cGT;\n", (ISMONO($2) ? 'M' : 'P'));
    >>
   |    "=" x2
    <<
        type_match("=", $0, $2);
        printf("\t%cEQ;\n", (ISMONO($2) ? 'M' : 'P'));
    >>
   ;
```

```
x2: x3 (("\+" << $0=$1; >> |"\-" << $0=$1; >>)  x3
    <<
        if (strcmp($1.text,"+")==0) {
            if (ISMONO($0)) printf("\tMADD;\n");
        else printf("\tPADD;\n");
            }
        else { if (ISMONO($0)) printf("\tMSUB;\n");
            else printf("\tPSUB;\n");
         }
    >>
    )*[$1]
    << $0 = $1; >>
  ;

x3: x4 (("\*" << $0=$1;>> |"/" << $0=$1; >>)    x4
    <<
        if (strcmp($1.text,"*")==0) {
            if (ISMONO($0)) printf("\tMMUL;\n");
            else printf("\tPMUL;\n");
            }
        else { if (ISMONO($0)) printf("\tMDIV;\n");
            else printf("\tPDIV;\n");
         }
    >>
    )*[$1] << $0=$1; >>
  ;

x4: x5
    {"^"    x5
    <<  if (ISMONO($0) && ISMONO($2)) {
            printf("\tMPOW\n");
        } else if (ISPOLY($0) && ISPOLY($2)) {
            printf("\tPPOW\n");
        } else {
        error("Types of operands to ^ don't match");
        }
    >>
    }[$1]
    <<
        $0 = $1;
    >>
  ;
x5: x6
    << $0 = $1; >>
    {"\.\." x6
    <<  if (!(ISMONO($0) && ISMONO($2))) {
            error("Operands of .. must be mono values");
        }
        printf("\tPRANGE;\n");
        $$ = POLY;
    >>
    }[$1]
  ;
```

```
x6: x7 << found = off; >> {"\[" x   "\]"
    <<  found = on;
        if (!ISMONO($2))
        error("%s should be mono valued",$2);
    >>
    }
    <<
        if (found == on) {
                    $0 = MONO;
                    printf("\tPSUBLD;\n");
                } else  $0 = $1;
    >>
  ;

x7: "sin"   x7
    << printf(ISMONO($2) ? "\tMSIN;\n" : "\tPSIN;\n");
       $0 = $2;
    >>
  | "cos"   x7
    << printf(ISMONO($2) ? "\tMCOS;\n" : "\tPCOS;\n");
       $0 = $2;
    >>
  | "tan"   x7
    << printf(ISMONO($2) ? "\tMTAN;\n" : "\tPTAN;\n");
       $0 = $2;
    >>
  | "floor" x7
    << printf(ISMONO($2) ? "\tMFLOOR;\n" : "\tPFLOOR;\n");
       $0 = $2;
    >>
  | "ceil"  x7
    << printf(ISMONO($2) ? "\tMCEIL;\n" : "\tPCEIL;\n");
       $0 = $2;
    >>
  | "\-"    x7
    <<printf(ISMONO($2) ? "\tMNEG;\n"   : "\tPNEG;\n");
      $0 = $2;
    >>
  | "\("    x   "\)"  << $0 = $2; >>
  | "left" x7
    << printf("\tPLEFT;\n");
    $0 = $2;
    >>
  | "right" x7
    << printf("\tPRIGHT;\n");
    $0 = $2;
    >>
  | "shuf" x7
    << printf("\tPSHUF;\n");
    $0 = $2;
    >>
  | "ishuf" x7
    << printf("\tPISHUF;\n");
    $0 = $2;
```

```
  >>
| mode    x7
  <<
      if (ISMONO($1) && !(ISMONO($2))) {
          printf("\tMAKEMONO;\n");
      } else if (!ISMONO($1) && ISMONO($2)) {
          printf("\tPCONST;\n");
      }
      $0 = $1; /* make type be mode */
  >>
| "\["    x   (","    x   )*  "\]"
  <<
  error("Yet to be implemented");
  $0 = POLY;
  >>
| WORD
  (("\("
      <<
          printf("\tMARK;\n");
      >>
  x ("," x)* "\)")
      <<
          printf("\tFRAME;\n\t%s();\n",$0.text);
          $$ = ((modes[lookup($0.text)] == MONOWD) ?MONO :POLY);
      >>
  |
      <<
          note = lookup($0.text);
          switch (modes[note]) {
          case MONOWD:
              switch (scope[note]) {
              case GLOBAL:
                  printf("\tMLD(%d);\n", offsets[note]);
                  break;
              case LOCAL:
                  printf("\tMFLD(%d);\n", offsets[note]);
              }
              $$ = MONO;
              break;
          case POLYWD:
              switch (scope[note]) {
              case GLOBAL:
                  printf("\tPLD(%d);\n", offsets[note]);
                  break;
              case LOCAL:
                  printf("\tPFLD(%d);\n", offsets[note]);
              }
              $$ = POLY;
              break;
          }
      >>
  )[$1]
| CONST
  << printf("\tMCONST(%s);\n", $1.text); $0=MONO;
```

```
    >>
  | "pi"
    << printf("\tMCONST(3.141592654);\n"); $0=MONO;
    >>
  | "e"
    << printf("\tMCONST(2.71828);\n"); $0=MONO;
    >>
  | "input"
    << printf("\tINPUT;\n"); $0=MONO;
    >>
  | "#"
    << printf("\tMCONST(POLYMAX);\n"); $0=MONO;
    >>
  ;<</* empty action for error */>>

#token  CONST   "[0-9][0-9]*"
#token  WORD    "[a-zA-Z_][A-Za-z0-9_]*"    <<;>>


<<
main()
{
    Attrib tmp;

    strcpy(tmp.text, "ick");
    ANTLRi(prog, tmp, stdin);
}


error(s, a, b, c, d)
char *s;
int a, b, c, d;
{
    fprintf(stderr, "Error Line %d -- ", lex_line);
    fprintf(stderr, s, a, b, c, d);
    fprintf(stderr, "\n");
}

type_match(s, a, b)
char *s;
Attrib a, b;
{
    if (ISMONO(a) != ISMONO(b)) {
        error("Type mismatch for operands to %s", s);
    }
}
>>
```

Appendix B: `ajl.h`

```
#include <ctype.h>
#define STKSIZ 128
#define CHAR_MAX 32
#define LOCAL 2
#define GLOBAL 1
#define on 1
#define off 0
#define MONOWD 1
#define POLYWD 2
extern char mempool[], temp[CHAR_MAX];
extern int memnext;
extern int string[];
extern int types[];
extern int scope[];
extern int modes[];
extern int offsets[];
extern int type, sp, g_ptr, l_ptr, frame_ptr, mg_cnt, pg_cnt, ml_cnt,
           pl_cnt, mode_flg, oldoffset, flag;
extern int found, note;
extern int frame_ptr_array[];
```

Appendix C: ajl_sup.h

```
/*  ajl_sup.h

    Support code for AJL programs running under unix:
    include file.

    May 1990, A. Jhaveri & H. Dietz
*/

#include    <stdio.h>
#include    <math.h>

#define IF  if (*(--msp))
#define WHILE   for (;;) {
#define DO  if (!(*(--msp))) break;

#define POLYMAX 16      /* Number of elements in a poly */

#define mono    double      /* Type of a mono value */

#define MSTKMAX 1000        /* Mono value workspace */
extern  mono    mstk[];
extern  mono    *msp;


#define MONOBOP(OP) { \
    msp[-2] = msp[-2] OP msp[-1]; \
    --msp; \
}

#define MONOP(OP)   { \
    msp[-1] = OP(msp[-1]); \
}

#define MONOUOP(OP) { \
    extern double OP(); \
 \
    msp[-1] = OP(msp[-1]); \
}


#define MPRINT  { \
    printf("%g\n", ((double) *(--msp))); \
}


#define MLT MONOBOP(<)
#define MLE MONOBOP(<=)
#define MNE MONOBOP(<>)
#define MGE MONOBOP(>=)
#define MGT MONOBOP(>)
#define MEQ MONOBOP(==)
```

```
#define MADD    MONOBOP(+)
#define MSUB    MONOBOP(-)

#define MMUL    MONOBOP(*)
#define MDIV    MONOBOP(/)

#define MNEG    MONOP(~)

#define MPOW    { \
    extern double pow(); \
 \
    msp[-2] = pow((double) msp[-2], (double) msp[-1]); \
    --msp; \
}

#define MSIN    MONOUOP(sin)
#define MCOS    MONOUOP(cos)
#define MTAN    MONOUOP(tan)

#define MFLOOR  { \
    extern double floor(); \
 \
    msp[-1] = floor((double) msp[-1]); \
}
#define MCEIL   { \
    extern double ceil(); \
 \
    msp[-1] = ceil((double) msp[-1]); \
}

#define MCONST(x)   *(msp++) = (x)

#define MLD(x)  *(msp++) = mstk[x]
#define MST(x)  mstk[x] = *(--msp)

#define MFLD(x) { *(msp++) = *(fp->f_mono + x); }
#define MFST(x) { *(fp->f_mono + x) = *(--msp); }
#define MRET    { \
    *(fp->f_mono) = msp[-1]; \
    psp = fp->f_poly; \
    msp = fp->f_mono + 1; \
    --fp; \
    return; \
}

#define INPUT   { \
    double d; \
 \
    scanf("%lf", &d); \
    *(msp++) = d; \
}

#ifndef M_PI
#define M_PI    3.141592654
```

```
#endif

#ifndef M_E
#define M_E 2.718281828
#endif

#define MAKEMONO    { \
    *msp = psp[-1]._[0]; \
    --psp; \
    ++msp; \
}

#define poly    struct _poly    /* Type of a poly value */
poly {
    mono    _[POLYMAX];
};

#define PSTKMAX 1000        /* Poly value workspace */
poly    pstk[];
poly    *psp;


#define POLYBOP(OP) { \
    register mono *q = &((--psp)->_[0]); \
    register mono *p = &(psp[-1]._[0]); \
    register mono *l = p + POLYMAX; \
 \
    do { \
        *p = (*p OP *(q++)); \
    } while (++p < l); \
}

#define POLYUOP(OP) { \
    extern double OP(); \
    register mono *p = &(psp[-1]._[0]); \
    register mono *l = p + POLYMAX; \
 \
    do { \
        *p = OP(*p); \
    } while (++p < l); \
}

#define POLYOP(OP)  { \
    register mono *p = &(psp[-1]._[0]); \
    register mono *l = p + POLYMAX; \
 \
    do { \
        *p = OP(*p); \
    } while (++p < l); \
}

#define forpoly(v)  for (v=0; v<POLYMAX; ++v)

#define PPRINT  pprint();
```

```
#define POLYIF { \
    register int v; \
 \
    forpoly(v) { \
        psp[-3]._[v] = (psp[-3]._[v] ? psp[-2]._[v] : psp[-1]._[v]); \
    } \
    psp -= 2; \
}

#define PLT POLYBOP(<)
#define PLE POLYBOP(<=)
#define PNE POLYBOP(<>)
#define PGE POLYBOP(>=)
#define PGT POLYBOP(>)
#define PEQ POLYBOP(==)

#define PADD    POLYBOP(+)
#define PSUB    POLYBOP(-)

#define PMUL    POLYBOP(*)
#define PDIV    POLYBOP(/)

#define PPOW    { \
    extern double pow(); \
    register mono *q = &((--psp)->_[0]); \
    register mono *p = &(psp[-1]._[0]); \
    register mono *l = p + POLYMAX; \
 \
    do { \
        *p = pow((double)*p, (double)*(q++)); \
    } while (++p < l); \
}

#define PSIN    POLYUOP(sin)
#define PCOS    POLYUOP(cos)
#define PTAN    POLYUOP(tan)

#define PNEG    POLYOP(~)

#define PFLOOR  { \
    extern double floor(); \
    register mono *p = &(psp[-1]._[0]); \
    register mono *l = p + POLYMAX; \
 \
    do { \
        *p = floor((double) *p); \
    } while (++p < l); \
}
#define PCEIL   { \
    extern double ceil(); \
    register mono *p = &(psp[-1]._[0]); \
    register mono *l = p + POLYMAX; \
 \
    do { \
```

```
        *p = ceil((double) *p); \
    } while (++p < l); \
}

#define PLEFT   { \
    register mono *p = &(psp[-1]._[0]); \
    register mono *l = p + (POLYMAX - 1); \
    register mono wrap = *p; \
 \
    do { \
        *p = *(p + 1); \
    } while (++p < l); \
    *p = wrap; \
}
#define PRIGHT  { \
    register mono *p = &(psp[-1]._[0]); \
    register mono *l = p + (POLYMAX - 1); \
    register mono wrap = *l; \
 \
    do { \
        *l = *(l - 1); \
    } while (p < --l); \
    *l = wrap; \
}
#define PSHUF   { \
    /* Perform SHUFfle using formula derived from that on page 489, \
       "Ultracomputers," by J. T. Schwartz, ACM Transactions on \
       Programming Languages and Systems, Oct. 1980. \
    */ \
    register int i; \
    poly temp; \
 \
    forpoly(i) { \
        register int j = i + i; \
 \
        if (j >= POLYMAX) j -= (POLYMAX - 1); \
        temp._[j] = psp[-1]._[i]; \
    } \
    psp[-1] = temp; \
}
#define PISHUF  { \
    /* Perform ISHUFfle using formula derived from that on page 489, \
       "Ultracomputers," by J. T. Schwartz, ACM Transactions on \
       Programming Languages and Systems, Oct. 1980. \
    */ \
    register int i; \
    poly temp; \
 \
    forpoly(i) { \
        register int j = i + i; \
 \
        if (j >= POLYMAX) j -= (POLYMAX - 1); \
        temp._[i] = psp[-1]._[j]; \
    } \
```

```
    psp[-1] = temp; \
}

#define PRANGE { \
    register mono *p = &(psp[0]._[0]); \
    register int i; \
 \
    ++psp; \
    forpoly(i) { \
        *(p++) = ((i * (msp[-1] - msp[-2])) / (POLYMAX-1)) + msp[-2]; \
    } \
    msp -= 2; \
}

#define PCONST  { \
    register mono *p = &(psp[0]._[0]); \
    register int i; \
 \
    ++psp; \
    --msp; \
    forpoly(i) { \
        *(p++) = *msp; \
    } \
}

#define PSUBST  { \
    psp[-1]._[(int) msp[-2]] = msp[-1]; \
    msp -= 2; \
}

#define PSUBLD  { \
    msp[-1] = psp[-1]._[(int) msp[-1]]; \
    --psp; \
}


#define PLD(x)  *(psp++) = pstk[x];
#define PST(x)  pstk[x] = *(--psp);

#define PFLD(x) { *(psp++) = *(fp->f_poly + x); }
#define PFST(x) { *(fp->f_poly + x) = *(--psp); }
#define PRET    { \
    *(fp->f_poly) = psp[-1]; \
    psp = fp->f_poly + 1; \
    msp = fp->f_mono; \
    --fp; \
    return; \
}

#define MAXFRAMES   256
#define frame   struct _frame
frame {
    mono    *f_mono;
    poly    *f_poly;
```

```
};

extern  frame   fstack[];
extern  frame   *fp;

#define MARK    { register mono *fp_mono = msp; \
                  register poly *fp_poly = psp;
#define FRAME     ++fp; fp->f_mono = fp_mono; \
                  fp->f_poly = fp_poly; }

extern  void    debug();
```

Appendix D: `ajl_sup.c`

```
/*  ajl_sup.c

    Support code for AJL programs running under unix:
    global data initialization and support functions.

    May 1990, A. Jhaveri & H. Dietz
*/

#include    "ajl_sup.h"

mono    mstk[MSTKMAX];
mono    *msp = &(mstk[0]);

poly    pstk[PSTKMAX];
poly    *psp = &(pstk[0]);

frame   fstack[MAXFRAMES];
frame   *fp = &(fstack[0]);

pprint()
{
    register int i, j;

    printf("\n");
    for (i=0; i<((POLYMAX+3)/4); ++i) {
        printf("%d:%g", i, ((double) psp[-1]._[i]));
        for (j=((POLYMAX+3)/4); j+i<POLYMAX; j+=((POLYMAX+3)/4)) {
            printf("\t%d:%g", j+i, ((double) psp[-1]._[j+i]));
        }
        printf("\n");
    }
    printf("\n");

    --psp;
}

void
debug(s)
char *s;
{
    register int i, j;

    printf("In function %s\n", s);
}
```

Appendix E: `sym_table.c`

```
/* symbol table manager for ajl */

#include "ajl.h"
#include "tokens.h"

char *text[STKSIZ];
char temp[CHAR_MAX];
int types[STKSIZ];
int scope[STKSIZ];
int modes[STKSIZ];
int offsets[STKSIZ];
int type, sp,g_ptr, l_ptr, flag,mode_flg,oldoffset,mg_cnt,pg_cnt,ml_cnt,pl_cnt;
int found, note;
int frame_ptr_array[STKSIZ];
int frame_ptr = 0;

int
lookup(s)
char *s;
{
    register int i = sp;

    while (--i >= 0) {
        if (strcmp(s, text[i]) == 0) return(i);
    }

    /* make dummy entry...  to recover from undefined var */
    return( enter("UndefinedVariable", KVAR, LOCAL, MONOWD) );
}

static char *
strsav(s)
register char *s;
{
    /* save a copy of the string s...
    */
    extern char *malloc();
    register char *p = malloc( strlen(s)+1 );

    strcpy(p, s);
    return(p);
}

int
enter(s, type, scop, mode, set)
char *s;
int type, scop, mode, set;
{
    types[sp] = type;
    scope[sp] = scop;
    modes[sp] = mode;
    text[sp] = strsav(s);
```

```
    offsets[sp] = set;
    return(sp++);
}
```