uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

u Ottawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Peng Fu
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Master of Computer Science
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Application Modeling Description Language for Reconfigurable Co-Processor Computing

TITRE DE LA THÈSE / TITLE OF THESIS

Voicu Groza
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

Amiya Nayak
CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

A. El Saddik

N. Santoro

Gary W. Slater
LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES /
DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

# Application Modeling Description Language

# For

# Reconfigurable Co-Processor Computing

By

Peng Fu

School of Information Technology and Engineering

University of Ottawa

800 King Edward Ave., Ottawa,

Ontario, Canada

September 2005

A Thesis submitted in conformity with the requirements for the degree of Master of Computer Science

*Your file Votre référence*
*ISBN: 0-494-11275-1*
*Our file Notre référence*
*ISBN: 0-494-11275-1*

# Abstract

Growing demands on functionality in today's electronic products are leading to an increasing shift towards developing in-system programmable hardware, to increase design flexibility. As a result, the development of embedded applications is becoming more and more important.

As part of the "Embedded Research Architectures for Co-Design Environments (ERACE) project, we developed the Application Description Language (ADL) which is a new kind of powerful language that enables embedded system users to model applications that can utilize the advantages provided by embedded systems, and its translator which translates ADL provided by the user into application source code. Note that in the traditional model, this is the source code that would be compiled and executed on a host processor. We then present it to the Just-In-Time compiler, which prepares the source code for the reconfiguration flow and application flow of the system.

The ADL project that included the Application Description Language, Translator, and a Destination Language are named "PACO v0.1". The Destination Language was chosen to be ANSI C language, although it can be any other high level programming language. The PACO v0.1 system was developed using LEX & YACC. Any application that looks for patterns in its input, or has an input or command language is suitable for development using LEX & YACC. When the user inputs instructions based on the PACO ADL grammar (syntax), the PACO translator will translate it into C source code, which will be sent to a profiler for further processing.

In PACO v0.1, we provide some features such as scientific calculations, logic algebra, matrix computations, and programming logic – control flow performance.

# Acknowledgements

I would like to thank my supervisors Dr. Voicu Groza and Dr. Amiya Nayak for inspiring my interest in Real-time Embedded System and Reconfigurable Co-processor Computing, and for guiding me all along through my study. Their precious suggestions and criticisms always helped me a lot.

Thanks to all the people, especially to Rami Abielmona, Mohammad El-Kadri, Mohammad Elbadri and Nizar Sakr, in our group meetings of Reconfigurable Co-processor Computing, for their suggestions to this work.

Finally, thank to all my family members, especially to my Dad and Mum, for supporting me throughout my education.

# Table of Contents

# List of Tables

# List of Figures

# Chapter One

# Introduction

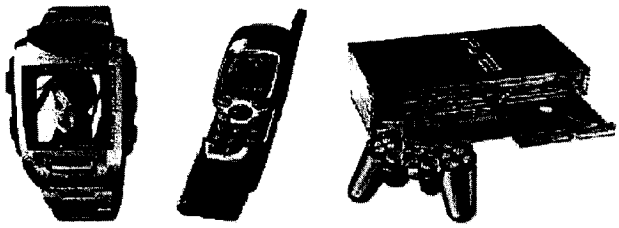
In this chapter, we give some concepts related to our research topic, such as embedded system, real-time system, reconfigurable co-processor computing, and real-time reconfiguration, etc. Those concepts will help us lead into our research project.

## 1.1 Embedded System

In today's world, computing systems are everywhere. There are millions of computing systems emerging every year. They range from desktop computers to laptops, from workstations to mainframes and servers. Meanwhile, there are billions of computing systems which are built for a very different purpose — Embedded System [27]. Embedded Systems are application-specific computing systems. They are embedded within larger electronic devices, repeatedly carrying out particular functions and completely running unrecognized by the user. It is quite difficult to define such embedded computing system precisely, even a simple embedded system. Different definitions for embedded systems can be found in the literature. In summary, an Embedded System is a specialized computer system that is part of a larger system or machine, which contains computer logic on a chip inside it not independently programmable by the end users, designed to perform a dedicated function. This definition is not perfect, but it may be as close as we will get.

There are varieties of common electronic devices that contain embedded systems. For example,

- Office automation: fax machines, copy machines, printers and scanners, etc.
- Business equipment: cash registers, curbside check-in, alarm systems, card readers, product scanners, and automated teller machines, etc.
- Consumer electronics: pagers, mobile phones, digital cameras, camcorders, VCR (videocassette recorders), DVD, portable video games, calculators, and PDA (personal digital assistants), etc.
- Home appliances: microwave ovens, smart telephones, answering machines, thermostats, home security systems, washing machines, and lighting systems, etc.
- Automobiles: transmission control, cruise control, fuel injection, antilock brakes, and active suspension, etc.



Casio Camera Watch | Nokia 7110 Browser Phone | Sony Playstation 2

Philips DVD Player | Philips TiVo Recorder

| | |
|---|---|
| Antilock breaks | Modems |
| Auto-focus cameras | MPEG decoders |
| Automatic teller machines | Network cards |
| Automatic toll systems | Network switches/routers |
| Automatic transmission | On-board navigation |
| Avionic systems | Pagers |
| Battery chargers | Photocopiers |
| Camcorders | Point-of-sale systems |
| Cell phones | Portable video games |
| Cell phone base stations | Printers |
| Cordless phones | Satellite phones |
| Cruise control | Scanners |
| Curbside check-in systems | Smart ovens/dishwashers |
| Digital cameras | Speech recognizers |
| Disk drives | Stereo systems |
| Electronic card readers | Teleconferencing systems |
| Electronic instruments | Television |
| Electronic toys/games | Temperature controllers |
| Factory control | Theft tracking systems |
| Fax machines | TV set-top boxes |
| Fingerprint identifiers | VCR's, DVD players |
| Home security systems | Video game consoles |
| Life-support systems | Video phones |
| Medical testing systems | Washer and dryer |

*Figure 1.1 A list of embedded systems*

Actually, nearly all the electronic devices either already have or will soon have a computing system embedded within them. Although embedded computers typically cost far less than desktop computers, their quantities are huge. Several billions of embedded microprocessor units are sold annually in recent years, compared to a few hundred millions desktop

microprocessor units. Therefore, new design methods are needed for the embedded systems that would emphasize on high-level tools and hardware-software tradeoffs, rather than low-level assembly language programming and logic design. In our project, we present a traditionally distinct field of software and hardware co-design in a new unified approach.

## 1.2 Real-time system

Many embedded systems can be characterized as *real-time*. A real-time system is one in which the correctness of the computations not only depends on their logical correctness, but also on the time at which the result is produced. In other words, a late answer is a wrong answer.

This window of opportunity imposes timing constraints on the operation of the machine. Applications with these kinds of timing constraints are considered real-time. In this case, the timing constraints are in the form of a period and deadline. The period is the amount of time between each iteration of a regularly repeated task. Such repeated tasks are called periodic tasks. The deadline is a constraint on the latest time at which the operation must complete.

A real-time system can be classified as either hard or soft. The distinction, however, is somewhat fuzzy. A hard real-time system is one in which one or more activities must never miss a deadline or timing constraint; otherwise, the system fails. Failure includes damage to the equipment, major loss in revenues, or even injury or death to users of the system. In contrast, a soft real-time system is one that has timing requirements, but occasionally missing them has negligible effects, as application requirements as a whole continue to be met.

## 1.3 Reconfigurable Co-processor System

The emergence of configurable computing gave rise to a new paradigm for a flexible solution for hardware-executable tasks of embedded system. However, this flexibility is overshadowed by the performance of such systems in terms of execution speed and reconfiguration time. From this, emerged the field of reconfigurable (or adaptive) computers [20, 21].

Reconfigurable computer architectures make use of a programmable-hardware device in order to perform those tasks, which would otherwise execute slower on the traditional fixed-hardware processor. Systems employing such architectures have already been implemented. These systems introduce *Run-Time Reconfigurable* (RTR) devices, which have the ability to change the functionality of the programmable-hardware while the system is executing a given flow. This introduces the concept of *Reconfigurable Processor Unit* (RPU) which is the programmable-hardware device, acting as a co-processor for the main fixed-hardware processor.

## 1.4 Real-Time Reconfiguration

Reconfigurable computing, using Field Programmable Gate Arrays (FPGAs) [13, 14], is emerging as an alternative to conventional Application-Specific Integrated Circuit (ASIC), which is a chip designed for a particular application and general- purpose processors, as opposed to the integrated circuits that control functions such as RAM in a PC.

Reconfigurable architectures can be customized in the post-fabrication stage for a wide range of applications, including multi-media, communications, networking, graphics and

cryptography, to achieve significantly higher performance over general or even special-purpose processor alternatives (such as digital signal processings). For convenience, we will use the term FPGA to refer to any type of reconfigurable data path, whether implemented using FPGAs or other forms of reconfigurable logic.

Recent developments in reconfigurable architectures have demonstrated that a tightly coupled reconfigurable co-processor with a general purpose CPU can achieve significant speedup on a general class of applications. The architecture also contains memory hierarchy and communication channels that connect the CPU, data path, and memory. The CPU can be used to implement control-intensive functions and system I/O, leaving the data path to accelerate computation-intensive parts of an application. This class of architecture defines a common, reusable platform for a wide range of applications, and potentially provides a better transistor utilization than a single CPU or a combined CPU and ASIC of comparable silicon area.

To exploit the potential performance gain provided by this class of architectures, a re-targetable framework has been developed which automatically compiles system-level applications specified in C to executables running on these platforms. At the core of the Nimble Compiler is a hardware/software-partitioning algorithm that partitions applications onto the CPU and the data path. As opposed to many co-synthesis algorithms that work at moderate to coarse granularities (such as task-level and function level) and extract task-level parallelism, the algorithm performs fine grain partitioning at the loop and basic block levels to exploit potential instruction-level parallelism to significantly accelerate important loops in the FPGAs.

There have been considerable research efforts in the co-design of conventional embedded hardware/software architectures. However, the partitioning problem for architectures containing reconfigurable FPGAs has a different requirement: it demands a two-dimensional partitioning strategy, in both spatial and temporal domains, while the conventional partitioning involves only spatial partitioning. Here, spatial partitioning refers to physical implementation of different functionality within different areas of the hardware resource. For dynamically reconfigurable architectures, besides spatial partitioning, the partitioning algorithm needs to perform temporal partitioning, meaning that the FPGA can be reconfigured at various phases of the program execution to implement different functionality. In this paper [14], the authors focussed on the temporal partitioning aspect. The input to the algorithm is a set of candidate loops for hardware, termed kernels, which have been extracted from the source application. Each loop has a software version and one or more hardware versions that represent different delay and area tradeoffs. The partitioning algorithm selects which loops to implement in the FPGA, and which hardware version of each loop should be used to achieve the highest application-level performance. Key issues with this approach are:

- The partitioning algorithm must effectively capture the dynamic reconfiguration costs. This is difficult as the number of reconfigurations for one kernel depends on which other kernels may go into the hardware.
- The algorithm must integrate compiler optimizations and hardware design space exploration into the hardware/software partitioning process.
- Partitioning must be guided by various forms of profiling information to accurately assess the tradeoffs between hardware and software implementations.

## 1.6 Current Languages for Embedded System Applications

Here, we present some of the available environments for hardware and software co-design that may be considered as software development platforms for Run-time Reconfigurable Systems. The pros and cons of these languages and tools, as listed in Table 1.1, depend on how fit they are for the hardware and software co-design.

| | Pros | Cons |
|---|---|---|
| **Handle C** | Handle- C [38] is a programming language designed for compiling programs into hardware images of FPGAs. It is a subset of C, extended with a few constructs for configuring the hardware device and to support generation of efficient hardware. | Both Handle-C and SA-C target hardware directly, so we cannot generate C code for further implementation. |
| **Single Assignment C** | Single Assignment C (SA-C) [41] is a variant of the C programming language that can be directly and intuitively mapped onto circuits, including FPGAs. | |

| | | |
|---|---|---|
| **System C** | System C [44] is a C++ class library that provides the necessary constructs to model system architecture including hardware timing, concurrency, and reactive behaviour that are missing in standard C++. | Actually System C is more focussed on the low-level hardware function design. |
| **Matlab** | MATLAB [12] provides a powerful interactive computing environment for numeric computation, visualization, and data analysis. Its wide range of commands, functions, and language constructs permits users to solve and analyze difficult computational problems from science and engineering without programming in a general-purpose language. | MATLAB is the most suitable language for our project, but it does not open C code to the end-users for free. |
| **Labview** | LabVIEW [35], or Laboratory Virtual Instrumentation Engineering Workbench, is a graphical programming environment based on the concept of data flow programming. This programming paradigm has been widely adopted by industry, academia, and research laboratories around the world as the standard for data acquisition and instrument control software. | It targets physical instruments rather than user interface applications, and it does not open C code. |

| | | |
|---|---|---|
| **UML** | UML [7][11][46] is a general-purpose notational language for specifying and visualizing complex software, especially large, object-oriented projects. UML provides us with the basic concept of modeling description language; gives some idea of building our ADL and methodology of developing. | The models of UML cannot always be exchanged among different tools without the loss of information. Moreover, it cannot provide us the C code for free as well. |

*Table 1.1 Languages for Embedded System*

There is another language named *Native Mapping Language* [36]. In many papers related to reconfigurable computing or co-processor implementation, the Native Mapping Language is often mentioned. In this section, a brief introduction to the Native Mapping Language to help understand its relationship with the reconfigurable embedded systems is provided.

*Native Mapping Language* (NML) is a structural language which, with a set of tools, assists the user when developing the *eXtreme Processing Platform* (XPP) applications whose architecture is used to overcome some of the deficiencies of microprocessor-based architectures and fine-grained reconfigurable devices.

The tools include a placer and router (**xmap**), a cycle accurate simulator (**xsim**) and a visualizer (**xvis**). In NML each configuration is defined as a structure of interconnections and

operations. NML supports primitives to program the *configuration manager* (CM). The CM is responsible for loading the configurations from one external memory to the local cache and for configuring the array. The **xmap** generates, from NML, the configuration data and the binary code to program the CM. The tool uses a force-based scheme to place the PEs used in the NML description. It can place and route complex examples in a few seconds. To hide the user from architecture details and to drastically shorten the time to program complex applications, a C compiler has been a focus of research and development.

NML is a very simply structured and easy to learn hardware description language, which makes all features of the XPP directly available to the developer. Algorithms should be formulated as data and control flow graphs - the nodes of the graph represent the processing elements, whereas the edges correspond to the connections. The NML compiler maps these graphs directly to the array of processing elements with its automatic place and route algorithms. NML allows the user to define Modules that can be instantiated on the array. This enables the programmer to use or define libraries for frequently used functions such as complex multiplication or matrix operations.

The sequence of configurations is defined in the form of a finite state machine. Results of the array can modify the configuration-flow by means of events. NML even allows to load registers on the array or to partially change configurations during operation.

# Chapter Two

# ERACE Project

An overview of our ERACE project [14], [19] is given in this chapter. It includes an introduction to Reconfigurable Co-processor Design and Implementation, investigation of the co-processor system architectures that have explored the use of reconfigurable computing to extend the processor's functionality and achieve better performance, and other similar research.

## 2.1 Reconfigurable Co-processor Design and Implementation

The *Reconfigurable Co-Processor* (RCoP) [42] is based on the interconnection of several *Field Programmable Gate Arrays* (FPGAs). There are two disadvantages of such systems. One is that the reconfiguration is slow and data transfer has to be done via I/O pins of the FPGA; the other one is that the communication delays introduced in order to achieve efficient parallelism among FPGAs incur significant overhead. The processing becomes inefficient if the FPGAs are used as a processing unit for multiple tasks or applications due to the added overhead of reprogramming the whole chip when only a portion of it needs to be changed. The solution is entirely implemented on a single Xilinx Virtex-II chip, a partially reconfigurable FPGA. Therefore, Run-Time Reconfigurable (RTR) has emerged.

Embedded systems are now able to exploit these features of FPGAs and they are capable of being used as programmable processing unit with the aid of a processor [37]. In our project, we use the MicroBlaze softcore processor from Xilinx. These two components serve as the backbone to an RTR environment that could be used as a rapid prototyping

platform of hardware and software co-design systems [2]. The whole project is named "Embedded Research Architectures for Co-Design Environments" (ERACE).

## 2.2 System Architecture

The system, as shown in Figure 2.1, consists of two flows: *Application Flow*, which is executed as a software process, and *Reconfiguration Flow*, which is generated by the JIT compiler in order to prepare the RPU to execute those sections of the code that JIT compiler evaluates to be more efficiently executed in hardware than in software. The reconfiguration flow makes use of Hardware Blocks (HBs), which are hardware representations of the tasks to be executed on the RPU, and which can be loaded onto and released from the RPU.

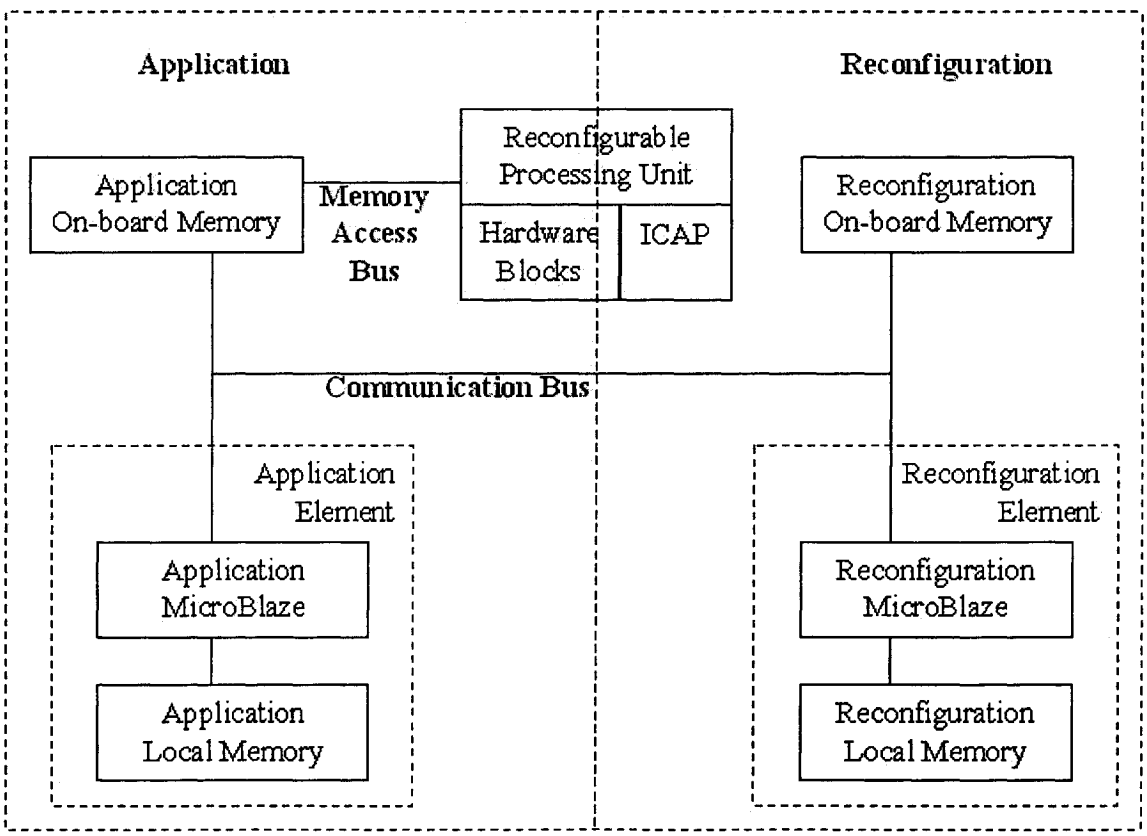



*Figure 2.1 Architecture of Run-time Reconfigurable System-on-Chip*

The *Application Element* houses *MicroBlaze*, which is the application microprocessor, and *Local Memory*. This is the execution unit for the application's task, which is to be executed in software (application software flow). The *Reconfigurable Processor Unit* is the execution unit for the application's tasks, which are to be executed in hardware, especifically on RTR Hardware Blocks (HBs). RPU is dynamically reconfigured with different HBs, as their corresponding bit streams are loaded onto or discarded from the RPU through the RPU's *Internal Configuration Access Port* (ICAP) by the *Reconfiguration Flow* running on the *Reconfiguration Element* which contains the reconfiguration microprocessor *MicroBlaze* and the *Local Memory*. This element executes the commands for RTR of RPU. The *On-Board Memory* houses two memory blocks: the *Application On-Board Memory* stores the user data and the instructions for *application flow*, whereas the *Reconfiguration On-Board Memory* stores the bit streams for the different tasks that are to be carried out in hardware on the RPU, as well as the instructions for the *reconfiguration flow*. The *Communication Bus* is used between the components. For our architecture, we are using the IBM *On-board Peripheral Bus* (OPB). The *Memory Access Bus* allows dedicated access to the application data in *Application On-Board Memory* by the RPU.

## 2.3 System Flow

The system flow is shown in Figure 2.2. The end users' input the script language named *Application Modeling Description Language* that we discussed before through lex.yy.exe file interface. The instructions are sent to the *PACO* translator to be interpreted into standard C language (ANSI C) – we call it *Application Source Code*. This is the main function of *PACO v0.1*. The *Just-In-Time Compiler* will process the Application Source Code that will make

use of our architecture, and prepare it for the *reconfiguration flow* and the *application flow* of the system.

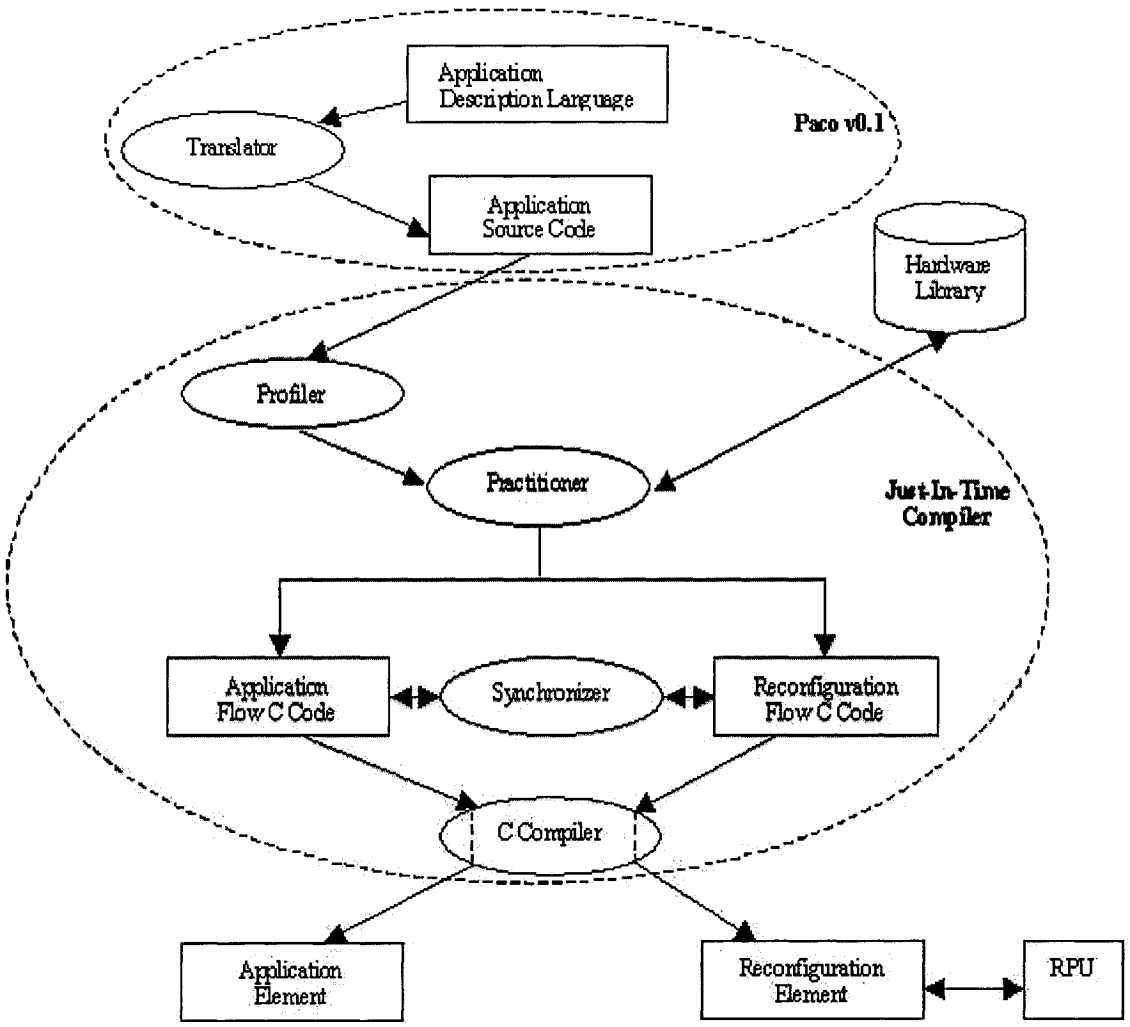



*Figure 2.2 System Flow Diagram*

The *profiler* provides us the functions and the number of times they are called, as well as the execution time of the functions in terms of clock cycles. In order to keep track of this data, a tree structure of these functions is created and updated. The structure also keeps track of the function call orders, as well as the dependency of functions upon each other. This information will be of use to the partitioner.

The *partitioner* checks if the particular function exists in our hardware library, and if it requires less clock cycles if it were to be executed in hardware. Based on this information, it decides which functions are to be executed using their Hardware Block (HB) representation and which should be executed in software. If a function is to be executed on the RPU, the partitioner calls a function of the same name sufficed with _RPU. When creating the reconfiguration source code, the partitioner keeps track of the HBs that will be present on the RPU at any given time slice. This is done by creating a map in memory that will designate an index to each HB with its corresponding address in memory.

The *synchronizer* ensures that the application and reconfiguration flows do not access areas of memory until the data that is to be read from the memory location is sane or until the memory location to be written to has been allocated correctly. To do this, the synchronizer needs to make sure that the register writes of the _RPU function do not occur before the HB is created, as well as not to access the result of the HB until it has finished executing. Once the RTR of the HB is finished, the application flow may then write the operand and result information to the HB's register. Then the application flow signals that the data is ready so that the HB may execute. Once the application flow needs to read the results, it has to wait until the HB has finished executing and acknowledge receipt of the results so the HB may terminate. Between these two critical points, the application flow can execute other branches of the tree created by the profiler that are independent of the operands and results of this matrix multiplication. The reconfiguration flow is synchronized with the application flow by ensuring that the HB is not enabled until the operand and result information has been written in its registers.

Currently, we can only execute functions whose hardware we have already implemented by recognizing the function name. A future version of this tool could be intelligent by teaching it to recognize certain software sequences so that it may realize their implementation in hardware by using neural networks.

# Chapter Three

## PACO v0.1

*PACO v0.1* is developed for Reconfigurable Co-processor Computing research purpose. Actually, PACO v0.1 is a software application with which the end users can access the Reconfigurable Co-processor Computing system, and program their instructions to make utilization of this embedded system. PACO v0.1 is a software system rather than a hardware driver. A regular driver just provides features to make hardware product run properly; it is not to provide functionalities of system control logic. Therefore, we define PACO as a software system, which consists of a couple of major aspects. The first one is the *Application Description Language*, which is the user input language; the second is a *Translator*, which will translate the ADL to the destination languages; and the final product from the translator is the *C source code*, which is for further implementations of the Just-In-Time Compiler.

In order to describe ADL vividly and for easy understanding, we use the concept of *Model* to define a set of equations on a set of data, which have been used for performing a computation. Models are a subset of ADL; users can put any model into PACO translator to get results. PACOv0.1 does not specify how the computations should be carried out. Instead, by using this system one can specify: models; algorithmic manipulation of such models; computerized simulation of systems; and manipulation of computerized files of such models.

A computation model describes a language and how the sentences (expressions, statements) of the language are executed by an abstract machine, based on which it uses a set of programming techniques to express solutions to the problems user wants to solve, and a set

of reasoning techniques to reason about programs to increase the confidence that they behave correctly and calculate their efficiency.

We use the modeling concept for the following reason. The model guarantees that the computations are done on (partial) data structures. All we need is to put our focus on functional programming and logic programming. Therefore, the design of ADL will be more structured and systematic.

In specification, when designing PACO v0.1, we divided the whole software system in four functional independent components:

- A model-oriented description language
- A user-oriented dialogue interface
- A translator and its library
- A C- source code file

A model-oriented description language is specifying the Application Description Language, which is a set of described models for the end-users to organize their instructions, in order to achieve the advantage over reconfigurable co-processor hardware computing. The design principle of ADL is easy to understand and use, by the end users.

A user-oriented dialogue interface is the ADL platform with which users can program their instructions. Currently, we use a DOS command line window style user interface in PACO v0.1. In a later version, we might design a friendly Graphic User Interface instead of DOS command line window.

A translator and its library is the kernel part of PACO v0.1. Obviously, translator is doing the translation job, which is from ADL built by the end users to a destination language that can be manipulated by the lower level software, which is Just-In-Time Compiler, into reconfigurable co-processor – the Reconfigurable Processing Unit. In our project, we chose ANSI C language as the destination language, although it can be any other high-level or low-level language such as assembly language. The choice of destination language depends on the compiler, which runs within the Just-In-Time compiler. The translator's library is a collection of functions for translating reference. When the end user submits any ADL model, the translator will consult the library for functions, and then compose the destination language by adding the functions in it. We will discuss this topic more specifically in the later chapters.

Finally, a C Source Code, which is our destination language, is generated. Actually we cannot interfere more with the C Source Code, because it is automatically generated. However, we can give the criteria of how to assemble the language in the translator part. Therefore, the translator is rather a template to form the matching rules between ADL and the destination languages.

From the designer's point of view, we know that our destination language should be an ANSI C Source Code, and then we translate ADL to it. The ADL structure should be either easy for the end users to manipulate, or effortless for translator to recognize and re-organize. So, in PACO, there is specification of the models; specification of the algorithmic manipulation the models; and specification of the simulation studies.

Rest of this thesis is devoted to the details of Application Description Language of PACO v0.1, the PACO Translator, and some experimental results to illustrate how PACO v0.1 works and what kind of role it plays in the whole project.

# Chapter Four

# Application Description Language

## 4.1 Language Design Principles

Programming languages [3] facilitate the mapping of abstractions from various domains of application into the computer representation of those abstractions. A programming language is thus a vehicle of communication between man and machine. Today there are dozens of high-level languages, and some commonly used high-level languages are BASIC, C, FORTAN and Pascal. The well-known high-level languages permit the implementation of algorithmic computer programs in notations that are both human oriented and machine translatable.

There are some high-level languages that have high abstraction compared to other high-level languages. We call them very high-level languages, which narrow the conceptual gap between man and machine by permitting a description of the problem to be solved; statement of the algorithmic solution is not explicitly required. The very high-level language translator automatically synthesizes an algorithmic program to solve the relationships among selected attributes of the system, and to present the solutions to the user in human oriented formats. By restricting the universe of denotation to a specific area of application, a high degree of structure and context can be incorporated into a very high level language, thus making the language easy to use and relatively easy to implement.

The high level languages [15] have certain technical characteristics, which facilitate the

specification of algorithmic processes. These characteristics include the syntactic structure, data types, data structures, operators, control mechanisms, I/O facilities, and storage management capabilities of the language. Very high-level languages incorporate certain characteristics that facilitate the models' description, and which include: [23]

- The structure of a model specifies the interrelationships among the independent and dependent variables of the model. The structural description includes specification of equations, number of dimensions, geometry, and the solution algorithm to be used, and the types of initial conditions and boundary conditions.

- Parameterization of a model permits specification of equation coefficients, boundary conditions, and initial conditions in the form of general arithmetic expressions, which can be arbitrary functions of the independent and dependent program variables. Thus, particular solutions to non-linear and time and space varying problems can be determined.

- The multiple return capabilities permit incremental updating of the model, thus permitting parametric solution runs in one batch processing cycle.

- Initialization control permits algorithmic manipulation of input data prior to the start of the simulation run. Termination control can assume various forms. In the simplest case, termination can be unconditional: for example, after 100 iterations of the integration algorithm. Conditional termination control can be phrased in terms of acceptable relative error between two solution iterations. Termination control can be

combined with the user extension capability to permit numerical solution of a two-point boundary value problem by iterative refinement of an initial value until two successive final values agree to within an acceptable error value.

- Output control specifies the solution output interval, and the format of the solution presentation.

- The user extension facility incorporation of user defined macros and subroutines in to simulation program. Macro definitions permit the definition of new statement forms in terms of existing statement forms. Thus, complex functions can be constructed from the elementary ones. User supplied algorithmic subroutines can be provided as the semantic definition of a new statement form. User supplied routines can also replace existing subroutines in the translator library.

In a nonprocedural language [17], the ordering of source statements is not related to the sequential order of the algorithmic code produced by the translator. Thus, source statements can appear in any order. In some cases, sorting routines are used to automatically order the source statements prior to translation.

## 4.2 Application Description Language

The Application Description Language (ADL) enables us to model applications [30] that can utilize the advantages provided by embedded system mentioned in Chapter 2. However, as the system will be dealing with adaptive applications, a system flow has to be generated in order to arrive at the embedded system, starting from ADL.

ADL is a language created to aid user in designing and creating application while hiding all hardware implementation and execution details. Here, we try to make the ADL provide a high-level abstracted UI (user interface) that is easy understood and use by the end users. They do not have to worry about exact steps of algorithm methods; just give the instructions to the system and then wait for the result. Therefore, the main advantage of the PACO v0.1 software system is that it can free the end users from complex program coding and excessive details of program languages. The ADL represents a collection of functions that can be modeled or programmed directly by the end users. In the current version of PACO, we provide some customized features in function formats, which can be called directly by the end users for corresponding implementations. Later on, the end users might be able to build their own functions by following some syntax rules provided by PACO designers. At present, we follow the syntax of standard C language (ANSI C) [22] rules when designing ADL.

Having a model for a software-hardware system prior to its construction is the most important issue. Good models are essential for communication among the individuals of the whole project and to guarantee architectural stability. We build models of complex systems because we cannot comprehend any such system in its entirety. As the complexity of systems increases, so does the importance of good modeling techniques. Having a strict description language standard is an essential factor. Such language must include:

**(1) Elements — fundamental units and semantics**

All the fundamental units in ADL are described as *elements* in the ADL specification. Everything from concrete language constructs such as variables, constants, expressions, strings, or other operators are referred to as elements.

**(2) Notation — visual rendering of model elements**

Under the operation of *notation*, elements can express a certain meaning. For instance, we can conjunct two or more elements by certain notation(s) to build up a meta-model. The meta-model is a data model used to describe designs expressed in ADL. The constituents of this model are the ADL elements that can be used to describe designs.

**(3) Rules — idioms of usage within the implementations**

With the *rules*, we can build up models now. Rules will make up a model by ranking and composing of meta-models for a particular functionality. Currently, we define the rules ourselves in advance, which means that the end users can utilize the existing models. In the later versions, the end users can specify their own rules.

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software. We look for techniques to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns, and frameworks. We also seek techniques to manage the complexity of systems as they increase in scope and scale.

Complexity will vary by application domain and process phase. One of the key motivations in the minds of the ADL developers is to create a set of semantics and notation that can adequately address all scales of architectural complexities across all domains.

The primary design goals of the ADL are as follows [18]:

- Serving and managing of Real-time Reconfigurable embedded system.
- Providing the users with a set of ready-to-use models to manipulate a certain operation.
- High extensibility mechanisms to extend the PACO software system.
- Independence of particular programming languages and development processes. The destination code can be in any kind of language such as C, C++, Java etc.
- Development for higher-level pattern concepts, which means we can develop a set of models to become a pattern.

The ADL fuses the concepts of a single, common, and widely usable description language for users of some methods. It pushes the envelope of what can be done with existing methods. It focuses on a standard description language, not a standard process. Although the ADL must be applied in the context of a process, it is our experience that different organizations and problems require different processes. Therefore, the efforts concentrated first on a common meta-model and second on a common notation.

The ADL specifies a description language that supports recurrence and embedding. We can call the same model within it for infinite times, and within one model we can call other model(s) as well. For example, when we do matrix multiplication, we can use the result of the previous matrix multiplications in the present processing. We can also use the result to do Fast Fourier Transform calculation. This feature provides, both to the user and RTR system, a flexible way to do the implementation.

## 4.3 Syntax (Grammar) of ADL in PACO v0.1

The syntax of a language defines how valid syntactically-correct programs can be written so that they can be executed by a machine (translator) [5]. The syntax is defined by a set of grammar rules. A grammar defines how to assemble "words" in to "sentences". Here, we call the sentence as "statement", "commands" or "expressions", which are all sentences in a programming language. By "words", we mean "tokens" that are the meaningful units in a programming language. Therefore, the grammar rules describe both tokens and statements. A statement is a sequence of tokens, while a token is a sequence of characters, which will be recognized by lexical analyzer and then produce a sequence of tokens. The same way, we can define the "parser" as a program that recognizes a sequence of tokens and produces a sequence of statement representation (see Figure 4.1). Normally the statements are represented as parse trees. The root is the statement itself; inter-nodes are expressions; and leaves are tokens (see Figure 4.2).

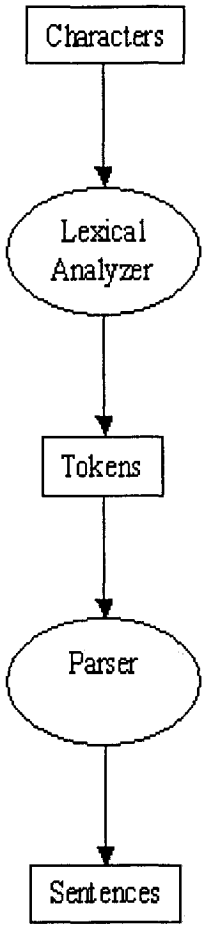



*Figure 4.1 Syntax flow*

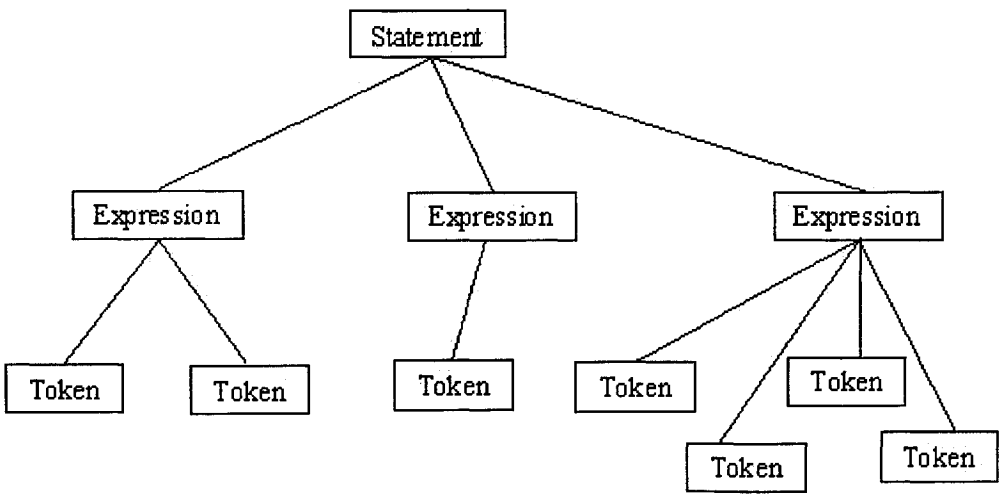



*Figure 4.2 Parse Tree*

To describe the grammar efficiently, we use Extended Backus-Naur Form (EBNF), which is a common notation to define grammars for programming languages. EBNF divides the symbols into 2 sets; one is terminal symbols (which are tokens), and the other is non-terminal symbols (which are sequences of tokens, presented by a grammar rule).

<non-terminal>::=<rule body>

We can simply define some grammar rules now.

- <digit>::=0|1|2|3|4|5|6|7|8|9|;

"<digit>" is defined to present one of the ten tokens 0-9; "|" means "or".

- <integer>::=<digit>{<digit>};

"<integer> is defined as the sequence of a <digit> followed by a zero or more <digit>s

- <x>; (a non-terminal x)
- <x>::=Body; (<x> is defined by Body)
- <x>|<y>; (either <x> or <y>)
- <x><y>; (the sequence <x> followed by <y>)
- {<x>}; (a sequence of zero or more occurrences of <x>)
- $\{<x>\}^{+}$; (a sequence of one or more occurrences of <x>)
- [<x>]; (zero or one occurrences of <x>)

The grammar rules can be used to either verify that a statement is legal, or to generate all possible statements. The set of all possible statements generated from a grammar and one non-terminal symbol is called a formal language.

The semantics of a language defines what a program does when it executes, and it should be simple and yet allow a programmer to reason about programs (correctness, execution time, and memory use). Here comes a question as to how can this be achieved for a practical language that is used to build complex systems? The kernel language approach can deal with that. The computation model of the kernel language is defined by defining how the constructs (statements) of the language manipulate (create and transform) the data structures (the entities) of the language. We also have to define a mapping scheme (translation) of full programming language into the kernel language. In our project, the practical language is ADL itself, and the kernel language is ANSI C. Actually, in the translating procedures translator will recognize each token and its meaning, its types, and other attributes. The grammar of ADL in PACO v0.1 is composed of 5 sets. Each set represents a certain kind of functions, which are made up by models.

### 4.3.1 Scientific Calculations

All kinds of the programming languages support scientific calculations. In ADL, we provide users with a set of functions for scientific calculations, including basic arithmetic and more complicated mathematics. In the future, we will add more features such as units of measure and conversions, and physical constants etc. to enlarge the scientific calculations features.

Numbers are represented by "<integer>" which means all set of numbers. We add "-" in the front to represent negative numbers. The system also accepts floating point numbers. The operators that can be used in current version of ADL are:

| | | |
|---|---|---|
| + | Addition | 59+268941; |
| - | Subtraction | 6589-81546.325; |

| | | |
|---|---|---|
| * | Multiplication | 24*52.2; |
| / | Division | 58/24.1 |
| ^ | Exponentiation | 76^8 |
| % | Modulo (finds reminder after division) | 8%7 |

The current version does not support compound operators such as "++", "--", "*=", "-=", and"+=".

***Meta-Model: element notation element***

Here, the *element* can be numbers, variables, and expressions. The *notation* can be any operator for scientific calculations. For example, if we want to add two elements, we use "A+B" to represent that. When we do a complex scientific calculation, we use recursive representation of the formula, such as *(element notation element) notation (element notation element)*, which means the element can be extended to a meta-model.

Example: 3049*459487^3-9

A+B*C-100.45

## 4.3.2 Logic Algebra

Algebra is a branch of mathematics in which symbols, usually letter of the alphabet, represent numbers or members of a specified set and are used to represent quantities and to express general relationships that hold for all members of the set. Logic algebra is an algebra in which elements have one of two values and the algebraic operations defined on the set are logical OR, a type of addition, logical AND, a type of multiplication, and logical NOT, a type of opposition. We denote letter of the alphabet as "[a-zA-Z]*". In ADL, the user can give

any definition to the parameters and use them as the operands of *AND*, *OR*, *NOT*. Any combination of such formula is accepted by the PACO translator.

The operators accepted by ADL are:

AND: &&

OR: ||

NOT: !

***Meta-Model: element notation element***

element && element

element || element

!element

Examples: A=0;

B=0;

C=1;

A&&B||!C;

## 4.3.3 Matrix Computations

Matrix computations are the essential part to prove that Real-Time Reconfigurable embedded system is far more efficient than the regular host computer systems. Matrix calculation part is not as exactly the same as in MATLAB; it uses a stricter style, for instance *x=[1,2,3]* is a row matrix (array); while *x=[1;2;3]* is a column one. Combining these notations, we may get

the matrix *x=[1,2,3;4,5,6]* which is a 2 by 3 matrix. Users may type, as well: *a=[1,2,3]; x=[a,4;1,2,3,4]*. The grammar of matrix computation is as simple as *c=a*x.*

***Meta-Model: element notation element***

Here the element refers to row matrix (array), column matrix (vector), and regular matrix. The notation could be matrix calculation operators, such as +, -, *, /.

Examples: a = [1, 0, -2; 0, 3, -1]

b = [0, 3; -2, -1; 0, 4]

c = a * b

d = c * c

## 4.3.4 Programming Logic – Control Flow Performance

Most programs involve repetition or *looping*. A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true. In ADL, we provide the users with three kinds of control flows: *for*, *do...while*, *while*.

***Meta-Model: notation elements***

for( ; ; )

do{ }while()

while() do{ }

We can compose a programming logic model by several different elements within a notation. We also support unlimited embedded loops, which means that there can be more than one loop within a loop.

Example: for (i=1; i<100; i=i+1) for (j=0; j<10; j=j+1) {x=x+1;};

while(i<10) {i=i+1;};

## 4.3.5 Fast Fourier Transform

A Fast Fourier Transform (FFT) [4] is an efficient algorithm to compute the Discrete Fourier Transform (DFT) and its inverse. It is of great importance to a wide variety of applications, from digital signal processing to solving partial differential equations to algorithms for quickly multiplying large integers.

***Meta-Model: element = notation (element)***

Here the notation for Fast Fourier Transform (FFT) is "R16SRFFT(parameter(s))" or "f(parameter(s))". Given below are the properties of the FFT method we use:

INPUT: float input[16], float output[16]

OUTPUT: none

EFFECTS: Places the 16 point fft of input in output in a strange order using 10 real multiplies and 79 real adds.

Re{F[0]}= out0

Im{F[0]}= 0

Re{F[1]}= out8

Im{F[1]}= out12

Re{F[2]}= out4

Im{F[2]}= -out6

Re{F[3]}= out11

Im{F[3]}= -out15

Re{F[4]}= out2

Im{F[4]}= -out3

Re{F[5]}= out10

Im{F[5]}= out14

Re{F[6]}= out5

Im{F[6]}= -out7

Re{F[7]}= out9

Im{F[7]}= -out13

Re{F[8]}= out1

Im{F[8]}=0

F[9] through F[15] can be found by using the formula

Re{F[n]}=Re{F[(16-n)mod16]} and Im{F[n]}= -Im{F[(16-n)mod16]}

FFT uses temporary variables to store intermediate computations in the butterflies, and this might speed things up. When the current version needs to compute a=a+b, and b=a-b, we do a=a+b followed by b=a-b-b. So practically everything is done in place, but the number of adds can be reduced by doing c=a+b followed by b=a-b. The algorithm behind this program is to find F[2k] and F[4k+1] eparately. To find F[2k] we take the 8 point Real FFT of x[n]+x[n+8] for n from 0 to 7. To find F[4k+1] we take the 4 point Complex FFT of exp(-2*pi*j*n/16)*{x[n] – x[n+8] + j(x[n+12]-x[n+4])} for n from 0 to 3.

Both formations are accepted by the translator. For the parameter(s) of Fast Fourier Transform, it can be a variable, such as "x", or an array, such as "[1, 2, 3, …, 16]".

Examples : x = [1, 2, 3, ..., 16] ;

y = R16SRFFT(x) ;

print(y) ;

or

y=f(1, 2, 3, ..., 16) ;

All the statements must be ended by ";", and meta-models support embedding and recurrence. We can embed a model or meta-model in another model or meta-model, and recurring a model or meta-model within another or more.

## 4.3 Present and Future

The ADL was developed for the embedded system software design, and system testing purpose. It addresses the needs of the user, as established by experience with the underlying methods on which it is based. The ADL builds upon similar semantics and notation from C language, Matlab, and other leading methods.

Although the ADL defines a precise language, it is not a barrier to future improvements in modeling concepts. The ADL can be extended without redefining the ADL core.

The ADL, in its current form, is expected to be the basis for embedded system tools, including those for simulation, and development environments. As interesting tool integrations are developed, implementation standards based on the ADL will become increasingly available.

The ADL has integrated many disparate ideas, and this integration will accelerate the use of object-orientation. Component-based development is an approach worth mentioning. It is synergistic with traditional object-oriented techniques. While reuse based on components is becoming increasingly widespread, this does not mean that component-based techniques will replace object-oriented techniques.

# Chapter Five

# Translator

The *Translator* in PACO v0.1 is a tool designed for ruling the mapping scheme from ADL that user inputs into C code file. First of all, the translator will read in the instructions from user input, recognize each symbols as token, and following the syntax which is given by the developer, reassemble each token into a new sentence – the C language statement, within C main function. The lex.yy.c file performs the tasks of the translator.

## 5.1 What is Translator?

Translator is a person who translates from one language to another, especially in writing as a profession. When we borrow this concept into computer science field, a translator or interpreter refers to a computer program that changes an instruction into a form that can be used directly by the computer, so that the instruction can be carried out at once. Here we extend the translator concept to describe the translator function in our project. The translator is a computer program that translates a computer program written in a computer language (called the *Source Language*) into an equivalent program written in another computer language (called the *Target Language*). In our case the Source Language is Application Description Language, and the Target Language is ANSI C language. As an important part of this translation process, the translator reports to its user the presence of errors in the source program. (See Figure 5.1)

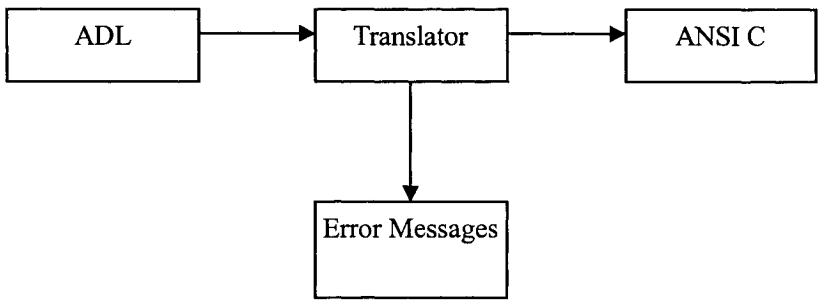



*Figure 5.1 PACO translator.*

## 5.2 Principles of Design

The PACO translator employs the "two stage" method – "*Front End*" and "*Back End*" [1]. The *Front End* translates ADL into an intermediate representation. The *Back End* is working with the internal representation to produce code in the output ANSI C. The Front End and the Back End can operate as separate passes, or the Front End can call the Back End as a subroutine, passing it the intermediate representation. This approach mitigates complexity, by separating the concerns of the Front End that typically revolve around language semantics and error checking, from the concerns of the Back End, which concentrates on producing output that is both efficient and correct. The advantage of this kind of system design is that it allows the user of single Back End for multiple source languages, and similarly allows the use of different Back Ends for different targets.

### 5.2.1 Translator's Front End

The front end consists of multiple phases itself:

- **Lexical Analysis**

  *Lexical Analysis* is breaking the source code text into small pieces -- tokens, each representing a single meaningful unit of the language. A finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called *Lexing*.

- **Syntax analysis**

  *Syntax Analysis* is for identifying syntactic structures of source code. It only focuses on the structure. In other words, it identifies the order of tokens and understands hierarchical structures in code. This phase is also called parsing.

- **Semantic analysis**

  *Semantic Analysis* is to recognize the *meaning* of program code and start to prepare for output. In that phase, type checking is done and most of compiler errors show up.

- **Intermediate language generation**

  In computer science, an intermediate language is the language of an abstract machine designed to aid in the analysis of computer programs. A compiler first translates a program into a form more suitable for code-improving transformations, as an intermediate step prior to generating code for the target machine.

### 5.2.2 Translator's Back End

While there are applications where only the Front End is necessary, a real translator hands the intermediate representation generated by the Front End to the Back End, which produces a functional equivalent program in the output language, which is ANSI C. This is done in multiple steps:

- **Compiler Analysis**

  This is the process to gather program information from the intermediate representation of the input source files. Typical analysis are variable define-use and use-define chain, data dependence analysis, alias analysis etc. Accurate analysis is the base for any compiler optimizations. The call graph and control flow graph are usually also built during the analysis phase.

- **Optimization**

  The intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion (source code of a different type that is written into the body of a program.); dead code elimination (used to reduce program size by removing code which can never be executed); constant propagation (the process of simplifying constant expressions at compile time), loop transformation (improving cache performance and effective use of parallel processing capabilities), register allocation (the process of multiplexing a

large number of target program variables onto a small number of CPU registers) or even auto parallelization.

- **Code generation**

  The transformed intermediate language is translated into the output language, usually the native machine language of the system, which is a set of instructions for a specific central processing unit, designed to be usable by a computer without being translated. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes.

## 5.3 Development Methodology – Lex & Yacc

*Lex and Yacc* [33] is a tool designed for writers of translators and compilers, although they are also useful for many other non-compiler applications. Actually, any application that looks for patterns in its input, or has an input or command language is better to be developed using Lex & Yacc. Furthermore, they allow for rapid application prototyping, easy modification, and simple maintenance of programs.

### 5.3.1 History of Lex & Yacc

Lex and Yacc were both developed at Bell Laboratories in the 1970s. Yacc was the first of the two, developed by Stephen C. Johnson [26]. Lex was designed by Mike Lesk [32] and Eric Schmidt [40] to work with Yacc. Both Lex and Yacc have been standard UNIX utilities since $7^{th}$ Edition UNIX. System V and older versions of BSD use the original AT&T versions, while the newest version of BSD uses flex and Berkeley Yacc. The articles written

by the developers remain the primary source of information on Lex and Yacc. The GNU Project of the Free Software Foundation distributes *bison*, a Yacc replacement; bison was written by Robert Corbett [8] and Richard Stallman [45]. The bison manual, written by Charles Donnelly [10] and Richard Stallman, is excellent, especially for referencing specific features. BSD and GNU Project also distribute *flex* (*F*ast *Lex*ical Analyzer Generator), "a rewrite of Lex intended to right some of that tool's deficiencies," according to its reference page. Flex was originally written by Jef Poskanzer and Vern Paxson [39] and Van Jacobson [25] have considerably improved it and Vern currently maintains it.

There are at least two versions of Lex and Yacc available for MS-DOS and OS/2 machines. MKS (Mortice Kern Systems Inc.), publishers of the MKS Toolkit, offers Lex and Yacc as a separate product that supports many PC C compilers. MKS Lex and Yacc come with a very good manual. Abraxas Software publishes PCYACC; a version of Lex and Yacc that comes with sample parsers for a dozen widely used programming languages.

### 5.3.2 Methodologies

Lex and Yacc help to write programs that transform structured input. Anything from a simple text search program that looks for patterns in its input file to a C compiler [1] that transforms a source program into optimized object code are included. Two tasks that occur over and over are, dividing the input into meaningful units, and then discovering the relationship among the units. For a text search program, the units would probably be lines of text, with a distinction between lines that contain a match of the target string and lines that do not. We call the units tokens. The division into units is *lexical analysis*, or *lexer*, or a *scanner* that can identify those tokens. The set of descriptions given to Lex is called a *lex specification*. The

token descriptions that lex uses are called *regular expressions*, extended versions of the familiar patterns used by the *grep* and *egrep* commands. Lex turns these regular expressions into a form that the lexer (short for lexical analyzer) can use to scan the input text extremely fast, independent of the number of expressions that it is trying to match. A lex lexer is almost always faster than a lexer written in C by hand.

When you write a *lex specification*, you create a set of patterns which lex matches against the input. Each time one of the patterns matches, the lex program invokes ADL that the user provided, which does something with the matched text. In this way, a lex program divides the input into strings, which are called tokens. Lex itself does not produce an executable program; instead it translates the lex specification into a file containing a C routine called *yylex()*. The program calls yylex() to run the lexer. Using regular C compiler, the user compiles the file that lex produced along with any other files and libraries that are needed.

The developer supplies the additional code beyond expression matching needed to complete the tasks, possibly including code written by other generators. The program that recognizes the expressions is generated in the general purpose programming language employed for the developer's program fragments. Thus, a high-level expression language is provided to write the string expressions to be matched while the user's freedom to write actions is unimpaired. This avoids forcing the user who wishes to use a string manipulation language for input analysis to write processing programs in the same and often-inappropriate string handling language.

LEX is a tool to generate and represent new language features, which can be added to

different programming languages, called "host language". Just as general-purpose languages can produce code to run on different computer hardware, Lex can write code in different host languages. The host language is used for the output code generated by Lex and also for the program fragments added by the user. Compatible run-time libraries for the different host languages are also provided. This makes Lex adaptable to different environments and different users. Each application may be directed to the combination of hardware and host language appropriate to the task, the user's background, and the properties of local implementations. At present, the only supported host language is C. Lex turns the user's expressions and actions (source) into the host general-purpose language; the generated program is named *yylex*. The yylex program will recognize expressions in a stream (inputs) and perform the specified actions for each expression as it is detected. See Figure 5.2.

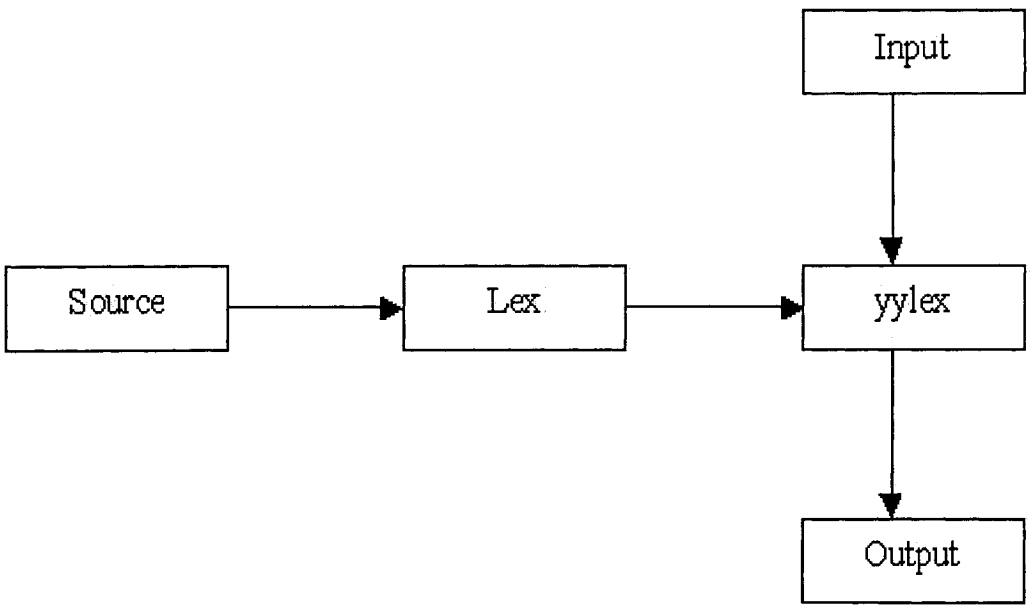



*Figure 5.2 An overview of Lex*

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a

lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context free grammars, but require a lower level analyzer to recognize input tokens. The context free grammar is a formal grammar in which every production rule is of the form V->w, where V is a non-terminal and w is a terminal and/or non-terminal. The term "context-free" comes from the fact that the non-terminal V can always be replaced by w, regardless of the context in which it occurs. Thus, a combination of Lex and Yacc is often appropriate. When used as a pre-processor for a later parser generator, Lex is used to partition the input stream, and the parser generator assigns structure to the resulting pieces. The flow of control in such a case (which might be the first half of a compiler, for example) is shown in Figure 5.3. Additional programs, written by other generators or by hand, can be added easily to programs written by Lex.

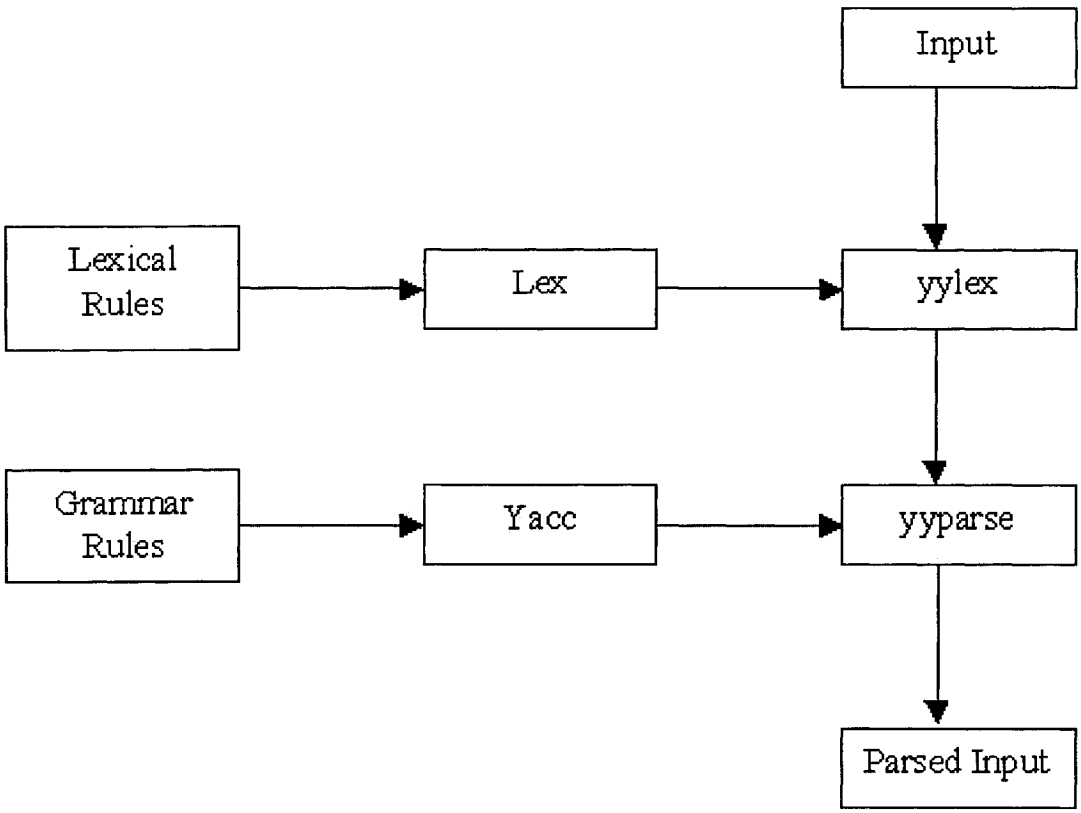



*Figure 5.3 Lex with Yacc*

Yacc users will realize that the name yylex is what Yacc expects its lexical analyzer to be named, so that the use of this name by Lex simplifies interfacing.

Lex generates a deterministic finite automaton from the regular expressions in the source. The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

Lex is not limited to source which can be interpreted on the basis of one character look ahead. For example, if there are two rules, one looking for `paco' and another for `pacocanada', and the input stream is `pacocanandaou', Lex will recognize 'paco' and leave the input pointer just before `canadaou'. Such backup is more costly than the processing of simpler languages.

As the input is divided into tokens, a program often needs to establish the relationship among the tokens. A C compiler needs to find the expressions, statements, declarations, blocks, and procedures in the program. This task is known as *parsing* and the list of rules that define the relationships that the program understands is a *grammar*. Yacc takes a concise description of a grammar and produces a C routine that can parse that grammar, a *parser*. The Yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and also detects a syntax error whenever its input does not match any of the rules. A Yacc parser is generally not as fast as a parser you could write by hand. However, the ease in writing and modifying the parser is invariably worth any speed loss. The amount of time a program spends in a parser is rarely enough to be an issue anyway.

## 5.3.3 LEX

The program lex generates a so-called "Lexer". This is a function that takes a stream of characters as its input, and whenever it seems a group of characters that match a key, takes a certain action.

Let's take a look at paco.l file.

```
%{
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include "paco.h"
#include "y_tab.h"

#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];


%}
```

In between the %{ and %} pair is included directly in the output program. We need this, because we use some functions later on, which are defined in pf.h.

```
D       [0-9]
S       [ \t]
NL      ((\n)|(\r\n))
SNL     ({S}|{NL})
EL      (\.\.\.)
BS      (\\)
CONT        ({EL}|{BS})
NOT         ((\~)|(\!))
POW         ((\*\*)|(\^))
EPOW        (\.{POW})
IDENT       ([a-zA-Z][_a-zA-Z0-9]*)
EXPON       ([DdEe][+-]?{D}+)
NUMBER      (({D}+\.?{D}*{EXPON}?)|(\.{D}+{EXPON}?)|(0[xX][0-9a-fA-F]+))

%x ML_COMMENT SL_COMMENT INCL USEIT
```

This section defines some "keys". Whenever the keys are encountered in the input, the rest of the line is executed.

Each section is separated using "%%", so the first line of the section will start right after %%.

```
<<EOF>>                         {
                                    if ( --include_stack_ptr < 0 ){
                                        yyterminate();
                                    }
                                    else{
                                        yy_delete_buffer( YY_CURRENT_BUFFER );
                                        yy_switch_to_buffer( include_stack[include_stack_ptr] );
                                    }
                                }
```

The "EOF" is a sign for terminate, inside gives the requirement when and what situation will terminate this program.

```
%%
"/*"                            BEGIN(ML_COMMENT);
<ML_COMMENT>[^*\n]*                 /* eat anything that's not a '*' */
<ML_COMMENT>"*"+[^*/\n]*            /* eat up '*'s not followed by '/'s */
<ML_COMMENT>\n                      ;
<ML_COMMENT>"*"+"/"                 BEGIN(INITIAL);

[/][/][^\n]*[\n]                ;

"while"                         return WHILE;
"for"                           return FOR;
"print"                         return PRINT;
"do"                            return DO;


{IDENT}                             {
                                        yylval.src = strdup(yytext);
                                        return IDENTIFIER;
                                    }

{NUMBER}                            {
                                        yylval.src = strdup(yytext);
                                        return NUMBER;
                                    }

">="                            return GE;
```

```
"<="                    return LE;
"=="                    return EQ;
"!="                    return NE;

"&&"                    return AND;
"||"                    return OR;
"!"                     return *yytext;

"?"                     return *yytext;

[-()<>=+*/;{}.,:\^\]\[]     {
                            return *yytext;
                        }

[ \t\r\n]+              ;   /* ignore whitespace */


.                       {
                            char* sz=malloc(yyleng+20);
                            strcpy(sz,">>>Unknown token: ");
                            strcat(sz,yytext);
                            yyerror(sz);

                            return LEXERROR;
                        }
%%
```

The above lex code is describing each kind of matches (tokens). For example, whenever lex reads in "while", it will give out "WHILE" for a match. A sequence of one or more characters from the group 0123456789 is written it shorter as [0-9]+. For letters, it can be presented by [a-zA-Z]*. The"+" signifies 1 or more matches; "*" means 0 or more matches. The notifications use in the paco.l, such as "while", "&&" etc, are pretty straightforward.

When an expression written as above is matched, lex executes the corresponding action. There is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus, the lex user who wishes to absorb the entire input, without producing any output must provide rules to match everything. It is normal that lex is being used with yacc. One may consider that actions are what *are* done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a

character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

Lex can handle ambiguous specifications. When more than one expression can match the current unput, lex chooses the longest match, and among rules which matched the same number of characters, the rule given first is preferred.

There are two steps in compiling a lex source program. First, the lex source must be turned into a generated program in the host general-purpose language. Then this program must be compiled and loaded, usually with a library of lex subroutines. The generated program is on a file named lex.yy.c. The I/O library is defined in terms of the C standard library.

### 5.3.4 YACC

YACC can parse input streams consisting of tokens with certain values. This clearly describes the relation YACC has with LEX; YACC has no idea what "inputs" are, and it needs reprocessed tokens. While we can write our own *Tokenizer*, we will leave that entirely up to LEX.

A note on grammars and parsers; YACC was used to parse input files for compilers: programs. Programs written in a programming language for computers are typically distinct – they have one meaning. As such, YACC does not cope with ambiguity.

```
%union {
    char* src;
    nodeType *nPtr;             /* node pointer */
};
```

```
%expect 0

%token <src> NUMBER
%token <src> IDENTIFIER
%token WHILE FOR   PRINT DO DOWHILE
%token ARRLIST ARGLIST PARA MATRIXLIST
%token EXPRLIST   LEXERROR SEP

%left ',' ';'
%left '='
%left GE LE EQ NE '>' '<'
%left OR
%left AND
%left '+' '-'
%left '*' '/'
%left UMINUS UPLUS
%left NOT
%right '^'
%left '(' '.' '{' '['

%type <nPtr> stmt expr stmt_list expr_list matrix_list
```

The above YACC code is the declarations section. Names representing tokens must be declared before use. Every name not defined in the declarations section is assumed to represent a non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

Of all the non-terminals, one is called the *start symbol*. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section. It is possible, and in fact desirable, to declare the start symbol explicitly in the declarations section using the %start keyword:

```
%start program
program          : opt_sep top_stmt opt_sep        { /*exit(0);*/ }
                 ;

top_stmt         : top_stmt opt_sep stmt           {
                                                       parserflag = nothing;
                                                       ex($3); freeNode($3);
                                                   }
                 | stmt                            {
```

```
                                        parserflag = nothing;
                                        ex($1); freeNode($1);
                                    }
        | LEXERROR                      { YYABORT; }
        ;

stmt    : expr ';'                      { $$ = opr(';', 1, $1); }
        | PRINT expr ';'                { $$ = opr(PRINT, 1, $2); }
        | expr '?'                      { $$ = opr(PRINT, 1, $1); }
        | WHILE '(' expr ')' stmt       { $$ = opr(WHILE, 2, $3, $5); }
        | DO '{' stmt_list '}' WHILE '(' expr ')' ';'
                                        { $$ = opr(DOWHILE, 2, $3,
$7); }
        | FOR '(' expr ';' expr ';' expr ')' stmt
                                        { $$ = opr(FOR, 4, $3, $5, $7,
$9); }
        | '{' stmt_list '}'             { $$ = $2; }
        ;

expr    : NUMBER                        { $$ = con($1); }
        | IDENTIFIER                    { $$ = id($1); }
        | '+' expr %prec UPLUS          { $$ = $2; }
        | '-' expr %prec UMINUS         { $$ = opr(UMINUS, 1, $2); }
        | '!' expr %prec NOT            { $$ = opr(NOT, 1, $2); }
        | expr '+' expr                 { $$ = opr('+', 2, $1, $3); }
        | expr '-' expr                 { $$ = opr('-', 2, $1, $3); }
        | expr '*' expr                 { $$ = opr('*', 2, $1, $3); }
        | expr '/' expr                 { $$ = opr('/', 2, $1, $3); }
        | expr '<' expr                 { $$ = opr('<', 2, $1, $3); }
        | expr '>' expr                 { $$ = opr('>', 2, $1, $3); }
        | expr '^' expr                 { $$ = opr('^', 2, $1, $3); }
        | expr AND expr                 { $$ = opr(AND, 2, $1, $3); }
        | expr OR expr                  { $$ = opr(OR, 2, $1, $3); }
        | expr GE expr                  { $$ = opr(GE, 2, $1, $3); }
        | expr LE expr                  { $$ = opr(LE, 2, $1, $3); }
        | expr NE expr                  { $$ = opr(NE, 2, $1, $3); }
        | expr EQ expr                  { $$ = opr(EQ, 2, $1, $3); }
        | '(' expr ')'                  { $$ = $2; }
        | IDENTIFIER '(' expr_list ')'  { $$ = opr(PARA, 2, id($1), $3); }
        | IDENTIFIER '(' ')'            { $$ = opr(PARA, 1, id($1)); }
                        | '[' ']'                       { $$ = opr(ARRLIST, 0); }
        | '[' matrix_list ']'           { $$ = opr(ARRLIST, 1, $2); }
        | IDENTIFIER '=' expr           { $$ = opr('=', 2, id($1), $3); }
        ;

expr_list   : expr                      { $$ = opr(EXPRLIST, 1, $1); }
            | expr_list ',' expr        { $$ = opr(EXPRLIST, 2, $3, $1); }
            ;

matrix_list : expr_list                 { $$ = opr(MATRIXLIST, 1, $1); }
            | matrix_list ';' expr_list { $$ = opr(MATRIXLIST, 2, $3, $1); }
                        ;

stmt_list   : stmt opt_sep              { $$ = $1; }
            | stmt_list stmt opt_sep    { $$ = opr(';', 2, $1, $2); }
```

```
                    ;

sep                 : ','
                    | ';'
                    | '\n'
                    | sep ','
                    | sep ';'
                    | sep '\n'
                    ;

opt_sep             : // empty
                    | sep
                    ;
```

The above section is the rule section, which is made up of one or more grammar rules. A grammar rule forms:

NON-TERMINAL: A_SEQUENCE_OF_ZERO_OR_MORE_NAMES_AND_LITERALS;

If there are several grammar rules with the same non-terminal, the vertical bar "|" is used to separate each grammar rule. With each grammar rule, the designer may associate actions to be performed each time the rule is recognized in the input processing. The actions may return values, and may obtain the values returned by previous actions. An action is an arbitraty C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "$" is used as a signal to YACC in this context. To return a value, the action normally sets "$$" to some value.

### 5.3.5 Implementations

To develop the translator, we use two files: one is named *paco.l*, which is for lexical analyzing by *FLEX*, and the other is *paco.y* for parsing by *BISON*. In file *paco.y*, we design the grammars that will be followed by the users to give the instruction that the reconfigurable co-processor will perform. In PACO v0.1, we try to simplify the grammar making it follow the C language style.

The two files will be working together under the calling of a file *BUILD.BAT* that is executing *FLEX/BISON*. We also develop some C code files to assist those two main files to enlarge the functionalities. We define the rules for memory allocation of each kind of parameters in *ctool.c* file. In *paco.c* file, there are some additional regulations, which are the supplementary of *paco.l*. Also, there are two head files -- *ctool.h* and *paco.h*. With those six files that we mentioned about, under the calling of *BUILD.BAT*, FLEX/BISON system will automatically generate the core part coding of the translator, which is a C language program named *lex.yy.c*. Here we use VC ++ as the default C compiler; but any other C compiler can be used. After compiling *lex.yy.c*, the result is an *.exe* file -- *lex.yy.exe* that will be used in the next step. Opening *lex.yy.exe* -- it says: "*start to input program from here: [type end in a new line to end]*". User may input code like the following example:

```
a=12;
b=34;
c=a+b;
d=a/b;
end
```

Then a file named *pf.c* is created. There are some other files that we create for library purposes when compiling *pf.c*. There is a file named *fastfouriertransform.c*, which is performing the Fast Fourier Transform function. Another very important library is *utl.c* and *utl.h* files. Those files provide data type transforming, matrix calculations, scientific calculations, logic algebras, array calculations, etc. All the processes can be done under the control of the same *BUILD.BAT*. The four C source or head files that we mentioned above can be implemented together in the Just-In-Time compiler for further processing. Actually, we code the features for Matrix and Fast Fourier Transform in function format in a separated C program file rather than in the main C program file *pf.c*.

The design methods are under the modeling language concept that we discussed earlier. Another advantage is in the future system expansion. Users may add any kind of functions later. It keeps open. More and more functions can be created for various purposes embedding in the core ADL.

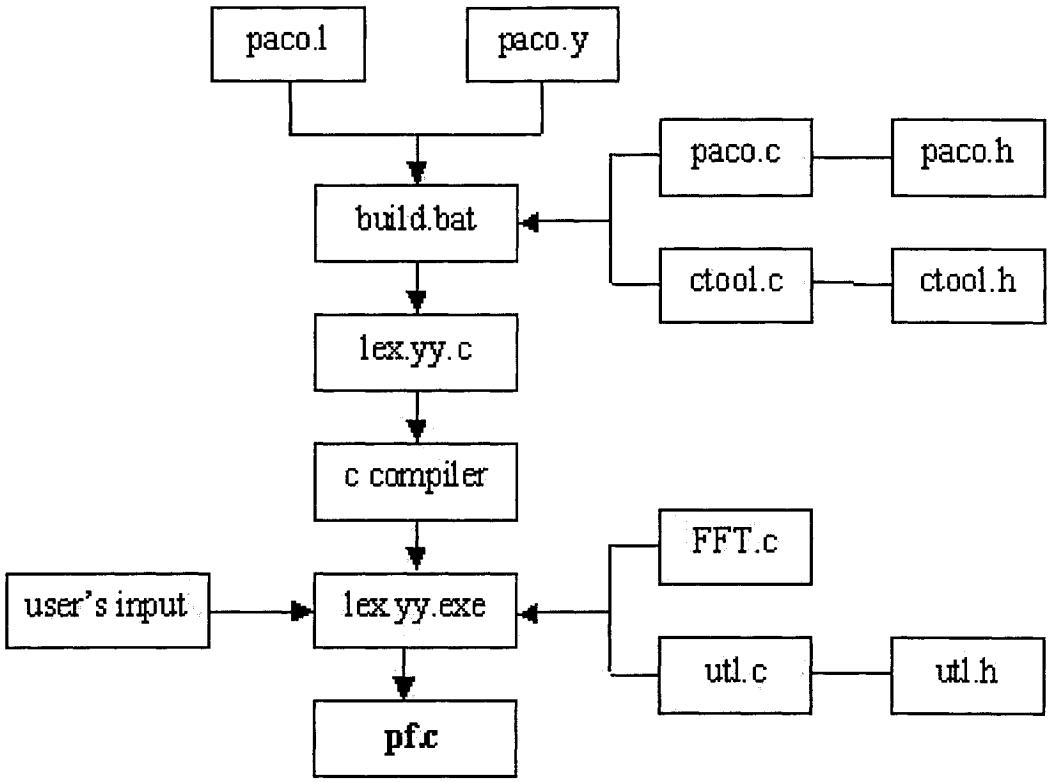



*Figure 5.4 PACO v0.1 software flow*

For the implementation, it is easy to do the program in C++ rather than C. One trick here is how to release memory automatically. That is the only part that gave the developers an headache when they coded them. In C++, Compiler knows when to release object's memory. However, in C, we need to take care of that by ourselves. There is a function named *realloc* for reallocating memory blocks, which is supported by VC++.

# Chapter Six

## Experimentation and Results

In this chapter, we will show some experimentation and results from PACO v0.1. The examples will follow the grammar we introduced in Chapter Three. We demonstrate the legal user inputs first in DOS command line window, and then show the final result from PACO v0.1, which is named *pf.c* file. The pf.c file is an ANSI C Source Code file that will be sent to Just-In-Time compiler to be profiled and portioned, then implemented in the Reconfigurable Processing Unit. To certify my work is correct, I use regular C compiler to run pf.c file, and the results are shown in each section.

## 6.1 Scientific Calculation

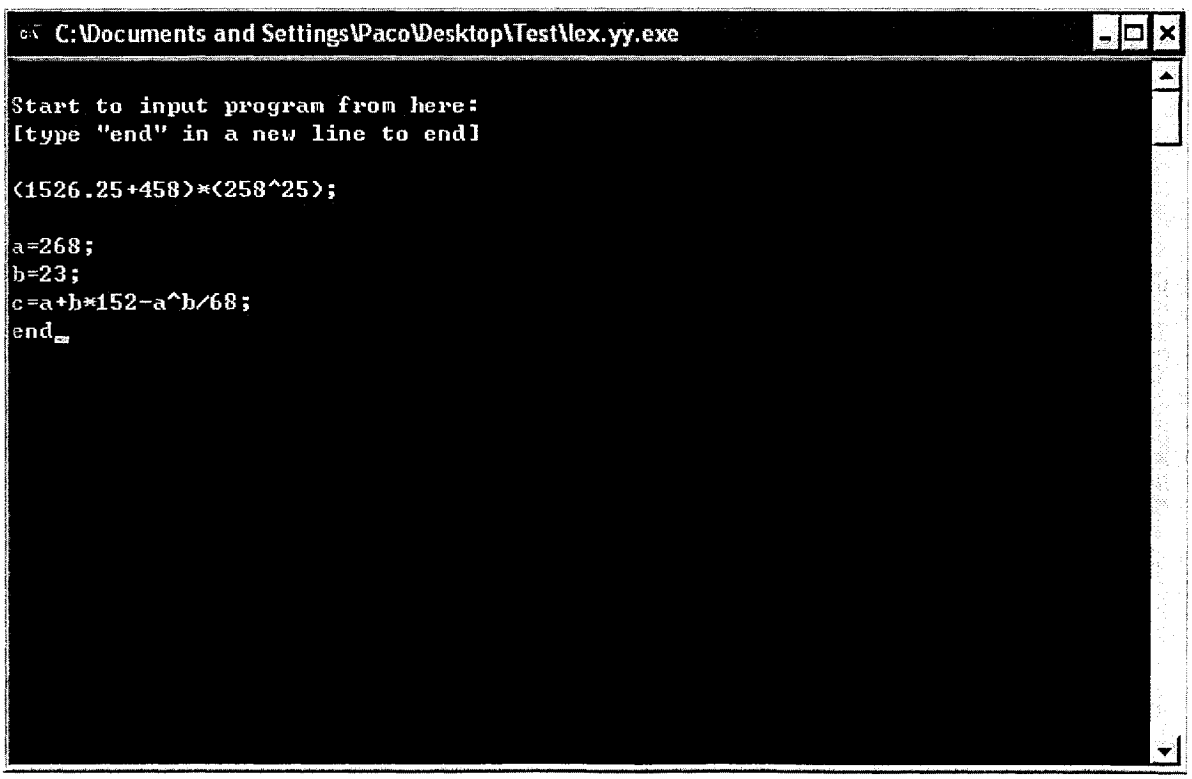



*Figure 6.1 Scientific Calculation A*

# pf.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "utl.h"

void main(void)
{

Var
*_Temp_17,*_Temp_16,*_Temp_15,*_Temp_14,*_Temp_13,*_Temp_12,*_Temp_11,*c,*_
Temp_10,*b,*_Temp_9,*a,*_Temp_8,*_Temp_7,*_Temp_6,*_Temp_5,*_Temp_4,*_Temp_
3,*_Temp_2,*_Temp_1,*pi;

_Temp_17=(Var *)malloc(sizeof(Var)); _Temp_17->tag_=typeNull;
_Temp_16=(Var *)malloc(sizeof(Var)); _Temp_16->tag_=typeNull;
_Temp_15=(Var *)malloc(sizeof(Var)); _Temp_15->tag_=typeNull;
_Temp_14=(Var *)malloc(sizeof(Var)); _Temp_14->tag_=typeNull;
_Temp_13=(Var *)malloc(sizeof(Var)); _Temp_13->tag_=typeNull;
_Temp_12=(Var *)malloc(sizeof(Var)); _Temp_12->tag_=typeNull;
_Temp_11=(Var *)malloc(sizeof(Var)); _Temp_11->tag_=typeNull;
c=(Var *)malloc(sizeof(Var)); c->tag_=typeNull;
_Temp_10=(Var *)malloc(sizeof(Var)); _Temp_10->tag_=typeNull;
b=(Var *)malloc(sizeof(Var)); b->tag_=typeNull;
_Temp_9=(Var *)malloc(sizeof(Var)); _Temp_9->tag_=typeNull;
a=(Var *)malloc(sizeof(Var)); a->tag_=typeNull;
_Temp_8=(Var *)malloc(sizeof(Var)); _Temp_8->tag_=typeNull;
_Temp_7=(Var *)malloc(sizeof(Var)); _Temp_7->tag_=typeNull;
_Temp_6=(Var *)malloc(sizeof(Var)); _Temp_6->tag_=typeNull;
_Temp_5=(Var *)malloc(sizeof(Var)); _Temp_5->tag_=typeNull;
_Temp_4=(Var *)malloc(sizeof(Var)); _Temp_4->tag_=typeNull;
_Temp_3=(Var *)malloc(sizeof(Var)); _Temp_3->tag_=typeNull;
_Temp_2=(Var *)malloc(sizeof(Var)); _Temp_2->tag_=typeNull;
_Temp_1=(Var *)malloc(sizeof(Var)); _Temp_1->tag_=typeNull;
pi=(Var *)malloc(sizeof(Var)); pi->tag_=typeNull;



CONST(_Temp_1,3.14159);
SET(pi,_Temp_1);
CONST(_Temp_2,1526.25);
CONST(_Temp_3,458);
ADD(_Temp_4,_Temp_2,_Temp_3);
CONST(_Temp_5,258);
CONST(_Temp_6,25);
```

```
POW(_Temp_7,_Temp_5,_Temp_6);
MUL(_Temp_8,_Temp_4,_Temp_7);
CONST(_Temp_9,268);
SET(a,_Temp_9);
CONST(_Temp_10,23);
SET(b,_Temp_10);
CONST(_Temp_11,152);
MUL(_Temp_12,b,_Temp_11);
ADD(_Temp_13,a,_Temp_12);
POW(_Temp_14,a,b);
CONST(_Temp_15,68);
DIV(_Temp_16,_Temp_14,_Temp_15);
SUB(_Temp_17,_Temp_13,_Temp_16);
SET(c,_Temp_17);

freevar(_Temp_17);
freevar(_Temp_16);
freevar(_Temp_15);
freevar(_Temp_14);
freevar(_Temp_13);
freevar(_Temp_12);
freevar(_Temp_11);
freevar(c);
freevar(_Temp_10);
freevar(b);
freevar(_Temp_9);
freevar(a);
freevar(_Temp_8);
freevar(_Temp_7);
freevar(_Temp_6);
freevar(_Temp_5);
freevar(_Temp_4);
freevar(_Temp_3);
freevar(_Temp_2);
freevar(_Temp_1);
freevar(pi);


}
```

After we compile pf.c with utl.c and utl.h files, we can get the result like:

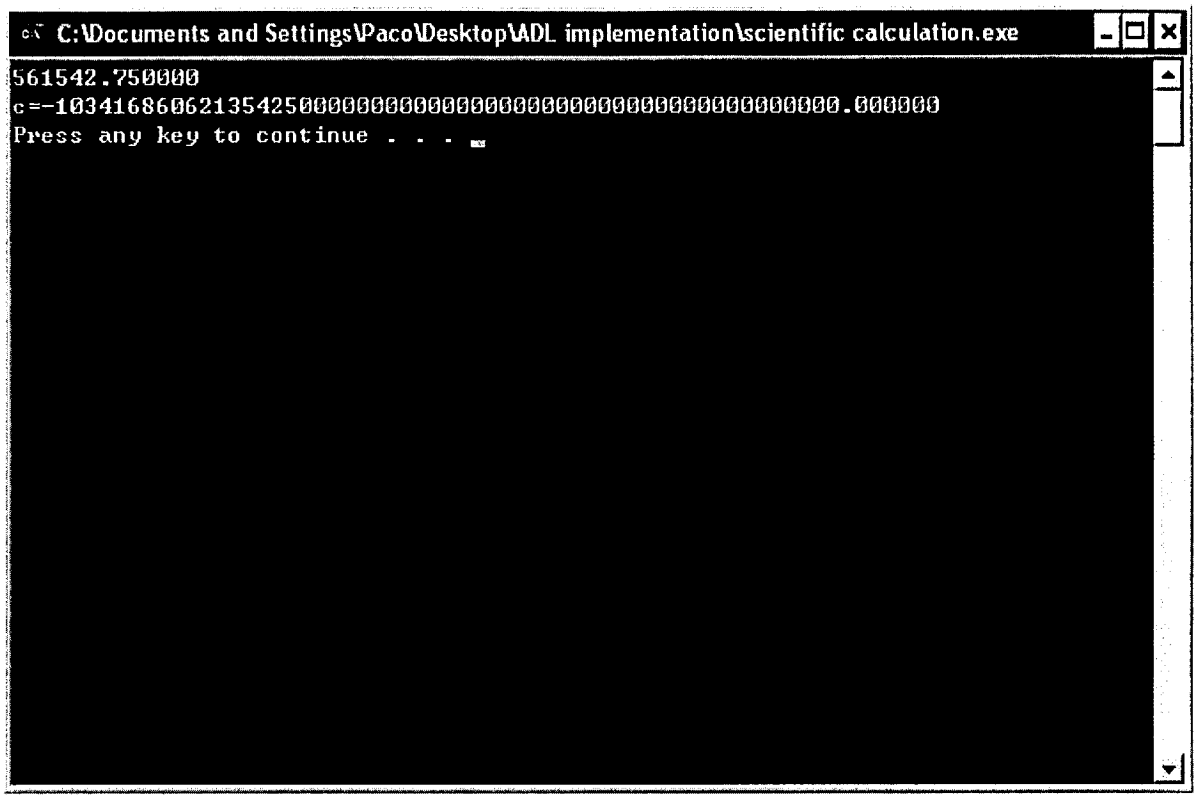



*Figure 6.2 Scientific Calculations B*

## 6.2 Logic Algebra

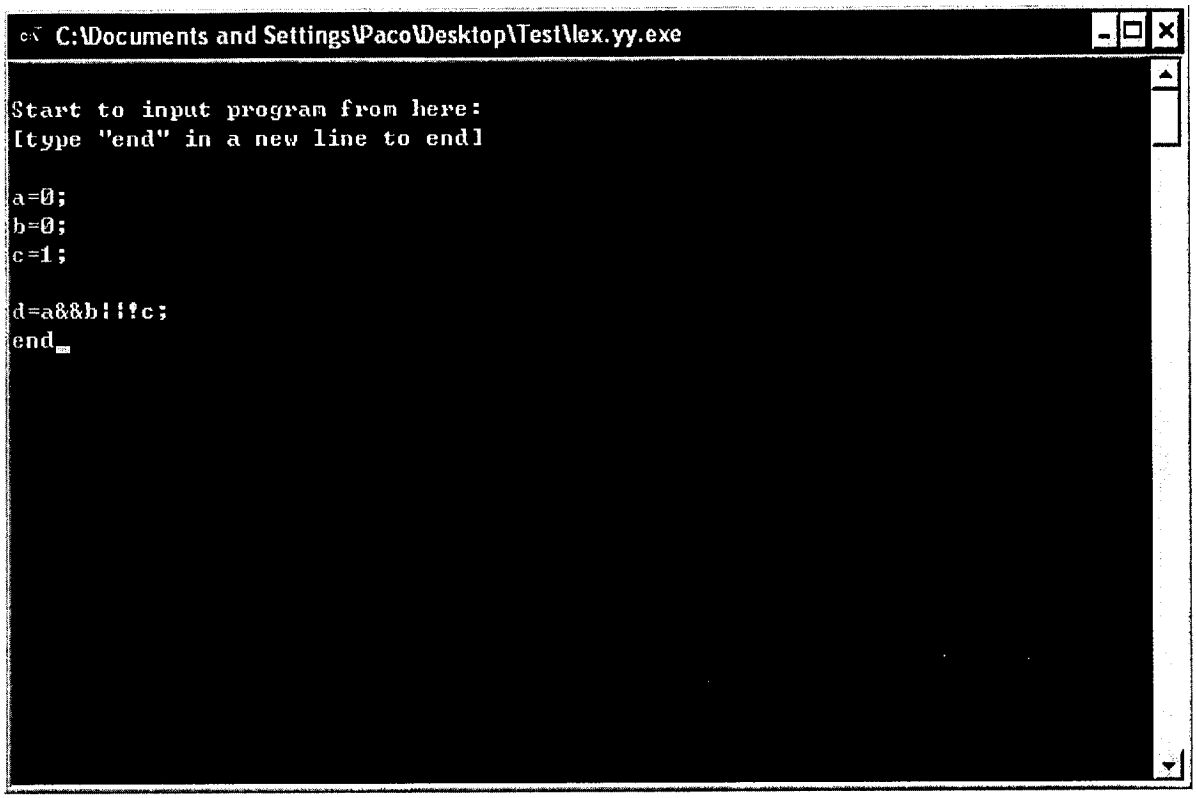



*Figure 6.3 Logic Algebra A*

## pf.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "utl.h"

void main(void)
{

Var
*_Temp_7,*_Temp_6,*_Temp_5,*d,*_Temp_4,*c,*_Temp_3,*b,*_Temp_2,*a,*_Temp_1,*
pi;
```

```
_Temp_7=(Var *)malloc(sizeof(Var)); _Temp_7->tag_=typeNull;
_Temp_6=(Var *)malloc(sizeof(Var)); _Temp_6->tag_=typeNull;
_Temp_5=(Var *)malloc(sizeof(Var)); _Temp_5->tag_=typeNull;
d=(Var *)malloc(sizeof(Var)); d->tag_=typeNull;
_Temp_4=(Var *)malloc(sizeof(Var)); _Temp_4->tag_=typeNull;
c=(Var *)malloc(sizeof(Var)); c->tag_=typeNull;
_Temp_3=(Var *)malloc(sizeof(Var)); _Temp_3->tag_=typeNull;
b=(Var *)malloc(sizeof(Var)); b->tag_=typeNull;
_Temp_2=(Var *)malloc(sizeof(Var)); _Temp_2->tag_=typeNull;
a=(Var *)malloc(sizeof(Var)); a->tag_=typeNull;
_Temp_1=(Var *)malloc(sizeof(Var)); _Temp_1->tag_=typeNull;
pi=(Var *)malloc(sizeof(Var)); pi->tag_=typeNull;

CONST(_Temp_1,3.14159);
SET(pi,_Temp_1);
CONST(_Temp_2,0);
SET(a,_Temp_2);
CONST(_Temp_3,0);
SET(b,_Temp_3);
CONST(_Temp_4,1);
SET(c,_Temp_4);
AND(_Temp_5,a,b);
NOT(_Temp_6,c);
OR(_Temp_7,_Temp_5,_Temp_6);
SET(d,_Temp_7);

freevar(_Temp_7);
freevar(_Temp_6);
freevar(_Temp_5);
freevar(d);
freevar(_Temp_4);
freevar(c);
freevar(_Temp_3);
freevar(b);
freevar(_Temp_2);
freevar(a);
freevar(_Temp_1);
freevar(pi);

}
```

After we compile pf.c with utl.c and utl.h files, we can get the result like:

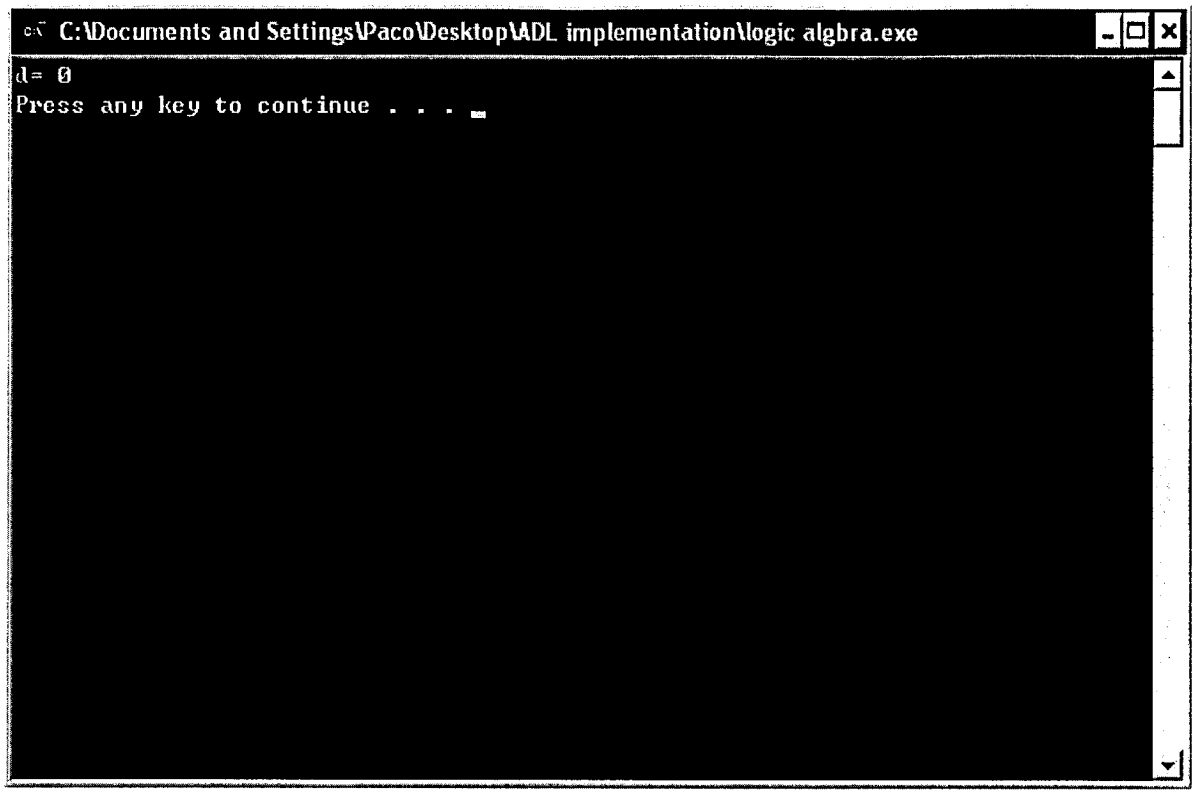



*Figure 6.4 Logic Algebra B*

## 6.3 Matrix Computations

In this example, we implement more than one calculation at the same time.

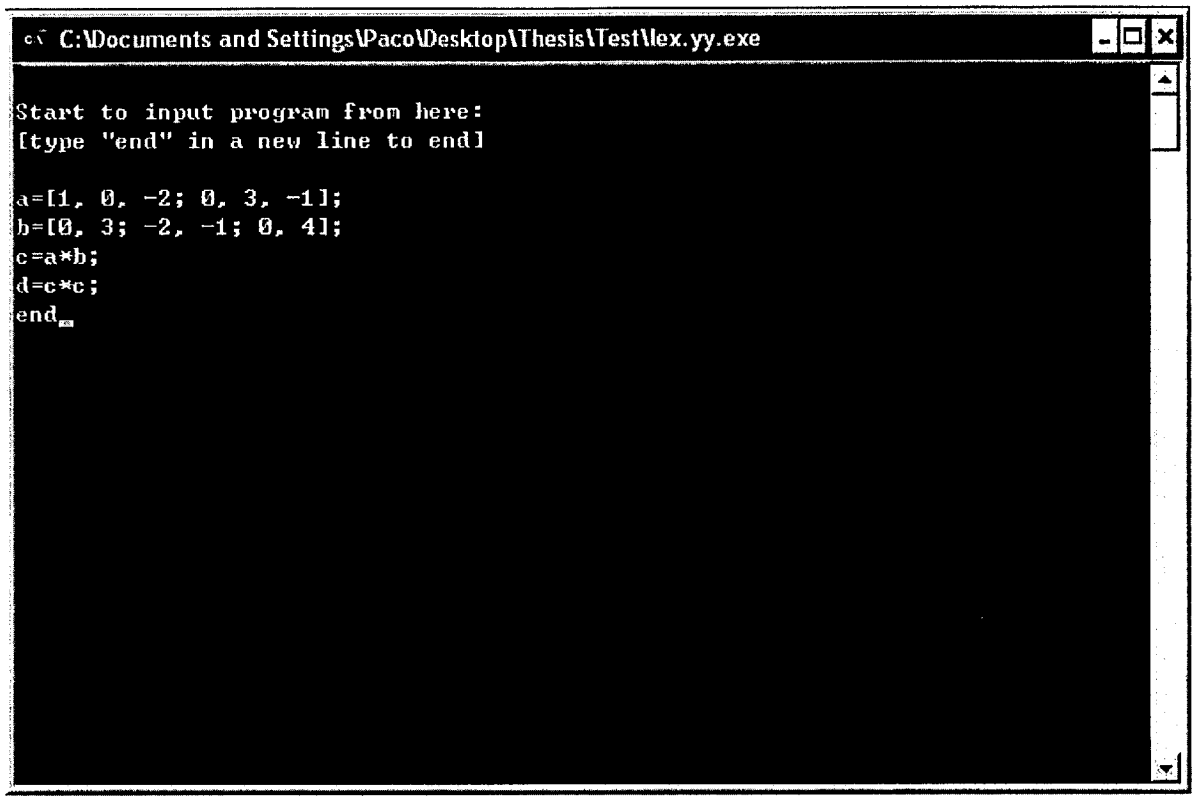



*Figure 6.5 Matrix Multiplication A*

# pf.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "utl.h"

void main(void)
{

Var
*_Temp_41,*d,*_Temp_40,*c,*_Temp_39,*_Temp_38,*_Temp_37,*_Temp_36,*_Temp_35
,*_Temp_34,*_Temp_33,*_Temp_32,*_Temp_31,*_Temp_30,*_Temp_29,*_Temp_28,*_Te
mp_27,*_Temp_26,*_Temp_25,*_Temp_24,*_Temp_23,*_Temp_22,*_Temp_21,*b,*_Tem
p_20,*_Temp_19,*_Temp_18,*_Temp_17,*_Temp_16,*_Temp_15,*_Temp_14,*_Temp_13,
*_Temp_12,*_Temp_11,*_Temp_10,*_Temp_9,*_Temp_8,*_Temp_7,*_Temp_6,*_Temp_5
```

```
,*_Temp_4,*_Temp_3,*_Temp_2,*a,*_Temp_1,*pi;

_Temp_41=(Var *)malloc(sizeof(Var)); _Temp_41->tag_=typeNull;
d=(Var *)malloc(sizeof(Var)); d->tag_=typeNull;
_Temp_40=(Var *)malloc(sizeof(Var)); _Temp_40->tag_=typeNull;
c=(Var *)malloc(sizeof(Var)); c->tag_=typeNull;
_Temp_39=(Var *)malloc(sizeof(Var)); _Temp_39->tag_=typeNull;
_Temp_38=(Var *)malloc(sizeof(Var)); _Temp_38->tag_=typeNull;
_Temp_37=(Var *)malloc(sizeof(Var)); _Temp_37->tag_=typeNull;
_Temp_36=(Var *)malloc(sizeof(Var)); _Temp_36->tag_=typeNull;
_Temp_35=(Var *)malloc(sizeof(Var)); _Temp_35->tag_=typeNull;
_Temp_34=(Var *)malloc(sizeof(Var)); _Temp_34->tag_=typeNull;
_Temp_33=(Var *)malloc(sizeof(Var)); _Temp_33->tag_=typeNull;
_Temp_32=(Var *)malloc(sizeof(Var)); _Temp_32->tag_=typeNull;
_Temp_31=(Var *)malloc(sizeof(Var)); _Temp_31->tag_=typeNull;
_Temp_30=(Var *)malloc(sizeof(Var)); _Temp_30->tag_=typeNull;
_Temp_29=(Var *)malloc(sizeof(Var)); _Temp_29->tag_=typeNull;
_Temp_28=(Var *)malloc(sizeof(Var)); _Temp_28->tag_=typeNull;
_Temp_27=(Var *)malloc(sizeof(Var)); _Temp_27->tag_=typeNull;
_Temp_26=(Var *)malloc(sizeof(Var)); _Temp_26->tag_=typeNull;
_Temp_25=(Var *)malloc(sizeof(Var)); _Temp_25->tag_=typeNull;
_Temp_24=(Var *)malloc(sizeof(Var)); _Temp_24->tag_=typeNull;
_Temp_23=(Var *)malloc(sizeof(Var)); _Temp_23->tag_=typeNull;
_Temp_22=(Var *)malloc(sizeof(Var)); _Temp_22->tag_=typeNull;
_Temp_21=(Var *)malloc(sizeof(Var)); _Temp_21->tag_=typeNull;
b=(Var *)malloc(sizeof(Var)); b->tag_=typeNull;
_Temp_20=(Var *)malloc(sizeof(Var)); _Temp_20->tag_=typeNull;
_Temp_19=(Var *)malloc(sizeof(Var)); _Temp_19->tag_=typeNull;
_Temp_18=(Var *)malloc(sizeof(Var)); _Temp_18->tag_=typeNull;
_Temp_17=(Var *)malloc(sizeof(Var)); _Temp_17->tag_=typeNull;
_Temp_16=(Var *)malloc(sizeof(Var)); _Temp_16->tag_=typeNull;
_Temp_15=(Var *)malloc(sizeof(Var)); _Temp_15->tag_=typeNull;
_Temp_14=(Var *)malloc(sizeof(Var)); _Temp_14->tag_=typeNull;
_Temp_13=(Var *)malloc(sizeof(Var)); _Temp_13->tag_=typeNull;
_Temp_12=(Var *)malloc(sizeof(Var)); _Temp_12->tag_=typeNull;
_Temp_11=(Var *)malloc(sizeof(Var)); _Temp_11->tag_=typeNull;
_Temp_10=(Var *)malloc(sizeof(Var)); _Temp_10->tag_=typeNull;
_Temp_9=(Var *)malloc(sizeof(Var)); _Temp_9->tag_=typeNull;
_Temp_8=(Var *)malloc(sizeof(Var)); _Temp_8->tag_=typeNull;
_Temp_7=(Var *)malloc(sizeof(Var)); _Temp_7->tag_=typeNull;
_Temp_6=(Var *)malloc(sizeof(Var)); _Temp_6->tag_=typeNull;
_Temp_5=(Var *)malloc(sizeof(Var)); _Temp_5->tag_=typeNull;
_Temp_4=(Var *)malloc(sizeof(Var)); _Temp_4->tag_=typeNull;
_Temp_3=(Var *)malloc(sizeof(Var)); _Temp_3->tag_=typeNull;
_Temp_2=(Var *)malloc(sizeof(Var)); _Temp_2->tag_=typeNull;
a=(Var *)malloc(sizeof(Var)); a->tag_=typeNull;
_Temp_1=(Var *)malloc(sizeof(Var)); _Temp_1->tag_=typeNull;
pi=(Var *)malloc(sizeof(Var)); pi->tag_=typeNull;
```

```
CONST(_Temp_1,3.14159);
SET(pi,_Temp_1);
CONST(_Temp_2,1);
ARR(_Temp_3,_Temp_2);
CONST(_Temp_4,0);
ARR(_Temp_5,_Temp_4);
ARH(_Temp_6,_Temp_3,_Temp_5);
CONST(_Temp_7,2);
NEG(_Temp_8,_Temp_7);
ARR(_Temp_9,_Temp_8);
ARH(_Temp_10,_Temp_6,_Temp_9);
CONST(_Temp_11,0);
ARR(_Temp_12,_Temp_11);
CONST(_Temp_13,3);
ARR(_Temp_14,_Temp_13);
ARH(_Temp_15,_Temp_12,_Temp_14);
CONST(_Temp_16,1);
NEG(_Temp_17,_Temp_16);
ARR(_Temp_18,_Temp_17);
ARH(_Temp_19,_Temp_15,_Temp_18);
ARV(_Temp_20,_Temp_10,_Temp_19);
SET(a,_Temp_20);
CONST(_Temp_21,0);
ARR(_Temp_22,_Temp_21);
CONST(_Temp_23,3);
ARR(_Temp_24,_Temp_23);
ARH(_Temp_25,_Temp_22,_Temp_24);
CONST(_Temp_26,2);
NEG(_Temp_27,_Temp_26);
ARR(_Temp_28,_Temp_27);
CONST(_Temp_29,1);
NEG(_Temp_30,_Temp_29);
ARR(_Temp_31,_Temp_30);
ARH(_Temp_32,_Temp_28,_Temp_31);
ARV(_Temp_33,_Temp_25,_Temp_32);
CONST(_Temp_34,0);
ARR(_Temp_35,_Temp_34);
CONST(_Temp_36,4);
ARR(_Temp_37,_Temp_36);
ARH(_Temp_38,_Temp_35,_Temp_37);
ARV(_Temp_39,_Temp_33,_Temp_38);
SET(b,_Temp_39);
MUL(_Temp_40,a,b);
SET(c,_Temp_40);
MUL(_Temp_41,c,c);
```

```
SET(d,_Temp_41);

freevar(_Temp_41);
freevar(d);
freevar(_Temp_40);
freevar(c);
freevar(_Temp_39);
freevar(_Temp_38);
freevar(_Temp_37);
freevar(_Temp_36);
freevar(_Temp_35);
freevar(_Temp_34);
freevar(_Temp_33);
freevar(_Temp_32);
freevar(_Temp_31);
freevar(_Temp_30);
freevar(_Temp_29);
freevar(_Temp_28);
freevar(_Temp_27);
freevar(_Temp_26);
freevar(_Temp_25);
freevar(_Temp_24);
freevar(_Temp_23);
freevar(_Temp_22);
freevar(_Temp_21);
freevar(b);
freevar(_Temp_20);
freevar(_Temp_19);
freevar(_Temp_18);
freevar(_Temp_17);
freevar(_Temp_16);
freevar(_Temp_15);
freevar(_Temp_14);
freevar(_Temp_13);
freevar(_Temp_12);
freevar(_Temp_11);
freevar(_Temp_10);
freevar(_Temp_9);
freevar(_Temp_8);
freevar(_Temp_7);
freevar(_Temp_6);
freevar(_Temp_5);
freevar(_Temp_4);
freevar(_Temp_3);
freevar(_Temp_2);
freevar(a);
freevar(_Temp_1);
freevar(pi);
```

}

After we compile pf.c with utl.c and utl.h files, we can get the result:


*Figure 6.6 Matrix Multiplication B*

## 6.4 Programming Logic -- Control Flow Performance

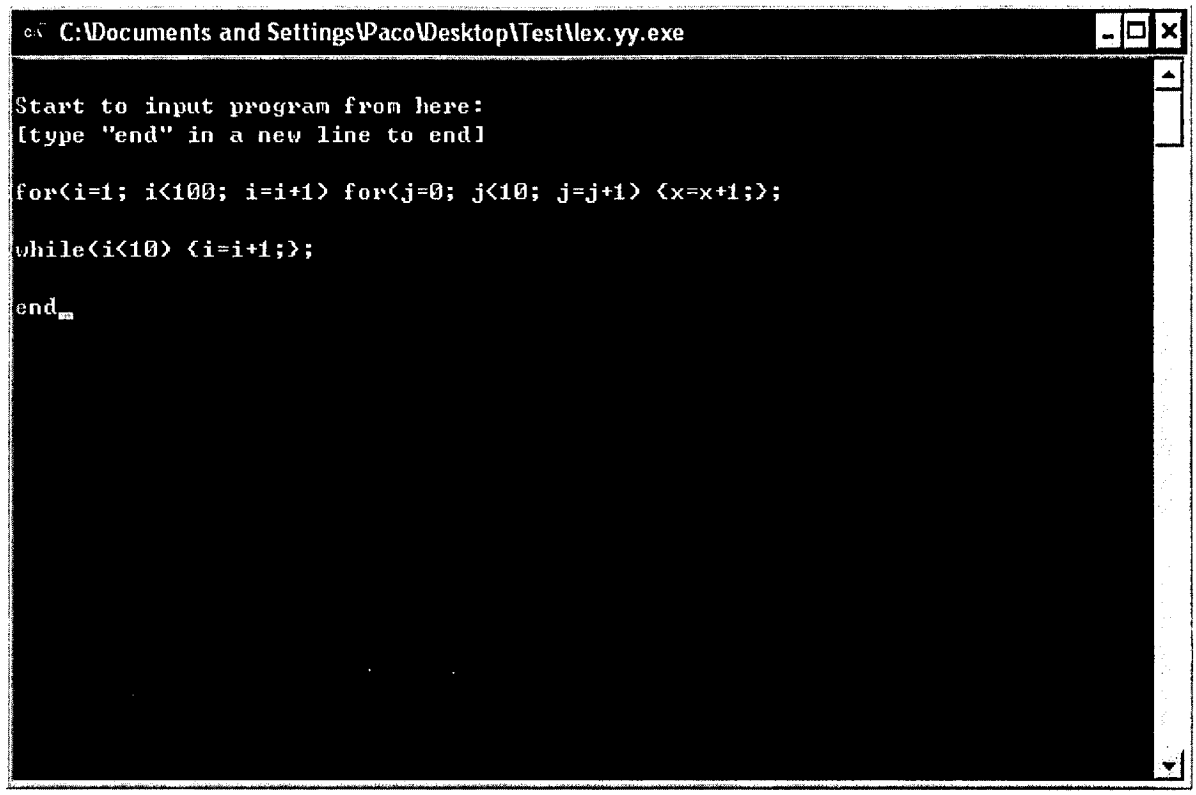



*Figure 6.7 Programming Logic A*

## pf.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "utl.h"

void main(void)
{

Var
*_Temp_17,*_Temp_16,*_Temp_15,*_Temp_14,*_Temp_13,*_Temp_12,*x,*_Temp_11,*_
Temp_10,*_Temp_9,*_Temp_8,*_Temp_7,*j,*_Temp_6,*_Temp_5,*_Temp_4,*_Temp_3,*
_Temp_2,*i,*_Temp_1,*pi;

_Temp_17=(Var *)malloc(sizeof(Var)); _Temp_17->tag_=typeNull;
_Temp_16=(Var *)malloc(sizeof(Var)); _Temp_16->tag_=typeNull;
```

```
_Temp_15=(Var *)malloc(sizeof(Var)); _Temp_15->tag_=typeNull;
_Temp_14=(Var *)malloc(sizeof(Var)); _Temp_14->tag_=typeNull;
_Temp_13=(Var *)malloc(sizeof(Var)); _Temp_13->tag_=typeNull;
_Temp_12=(Var *)malloc(sizeof(Var)); _Temp_12->tag_=typeNull;
x=(Var *)malloc(sizeof(Var)); x->tag_=typeNull;
_Temp_11=(Var *)malloc(sizeof(Var)); _Temp_11->tag_=typeNull;
_Temp_10=(Var *)malloc(sizeof(Var)); _Temp_10->tag_=typeNull;
_Temp_9=(Var *)malloc(sizeof(Var)); _Temp_9->tag_=typeNull;
_Temp_8=(Var *)malloc(sizeof(Var)); _Temp_8->tag_=typeNull;
_Temp_7=(Var *)malloc(sizeof(Var)); _Temp_7->tag_=typeNull;
j=(Var *)malloc(sizeof(Var)); j->tag_=typeNull;
_Temp_6=(Var *)malloc(sizeof(Var)); _Temp_6->tag_=typeNull;
_Temp_5=(Var *)malloc(sizeof(Var)); _Temp_5->tag_=typeNull;
_Temp_4=(Var *)malloc(sizeof(Var)); _Temp_4->tag_=typeNull;
_Temp_3=(Var *)malloc(sizeof(Var)); _Temp_3->tag_=typeNull;
_Temp_2=(Var *)malloc(sizeof(Var)); _Temp_2->tag_=typeNull;
i=(Var *)malloc(sizeof(Var)); i->tag_=typeNull;
_Temp_1=(Var *)malloc(sizeof(Var)); _Temp_1->tag_=typeNull;
pi=(Var *)malloc(sizeof(Var)); pi->tag_=typeNull;


CONST(_Temp_1,3.14159);
SET(pi,_Temp_1);
for(
CONST(_Temp_2,1),
SET(i,_Temp_2),_Temp_2;
CONST(_Temp_3,100),
LES(_Temp_4,i,_Temp_3),
ToBool(_Temp_4);
CONST(_Temp_5,1),
ADD(_Temp_6,i,_Temp_5),
SET(i,_Temp_6),_Temp_6){
for(
CONST(_Temp_7,0),
SET(j,_Temp_7),_Temp_7;
CONST(_Temp_8,10),
LES(_Temp_9,j,_Temp_8),
ToBool(_Temp_9);
CONST(_Temp_10,1),
ADD(_Temp_11,j,_Temp_10),
SET(j,_Temp_11),_Temp_11){
CONST(_Temp_12,1);
ADD(_Temp_13,x,_Temp_12);
SET(x,_Temp_13);}
}

while(
CONST(_Temp_14,10),
```

```
LES(_Temp_15,i,_Temp_14),
ToBool(_Temp_15))
{
CONST(_Temp_16,1);
ADD(_Temp_17,i,_Temp_16);
SET(i,_Temp_17);}

freevar(_Temp_17);
freevar(_Temp_16);
freevar(_Temp_15);
freevar(_Temp_14);
freevar(_Temp_13);
freevar(_Temp_12);
freevar(x);
freevar(_Temp_11);
freevar(_Temp_10);
freevar(_Temp_9);
freevar(_Temp_8);
freevar(_Temp_7);
freevar(j);
freevar(_Temp_6);
freevar(_Temp_5);
freevar(_Temp_4);
freevar(_Temp_3);
freevar(_Temp_2);
freevar(i);
freevar(_Temp_1);
freevar(pi);

}
```

After we compile pf.c with utl.c and utl.h files, we can get the result:

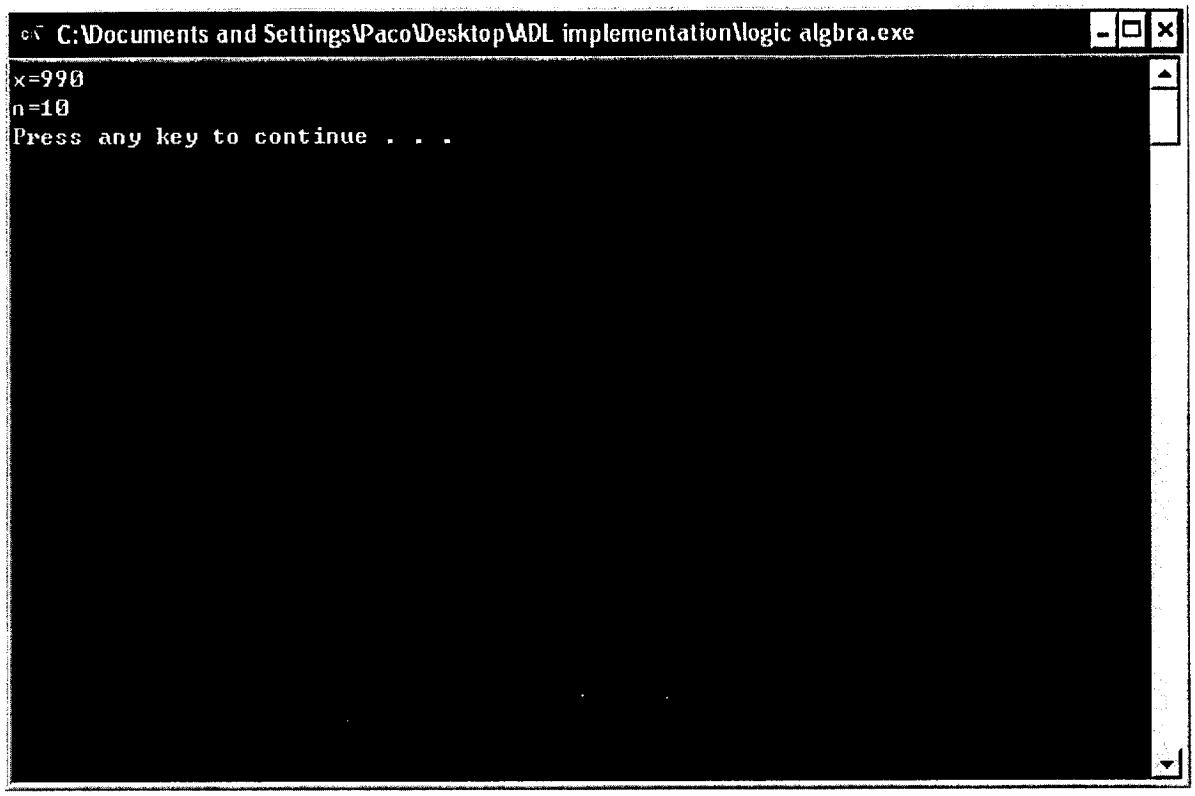



*Figure 6.8 Programming Logic B*

## 6.5 Fast Fourier Transform

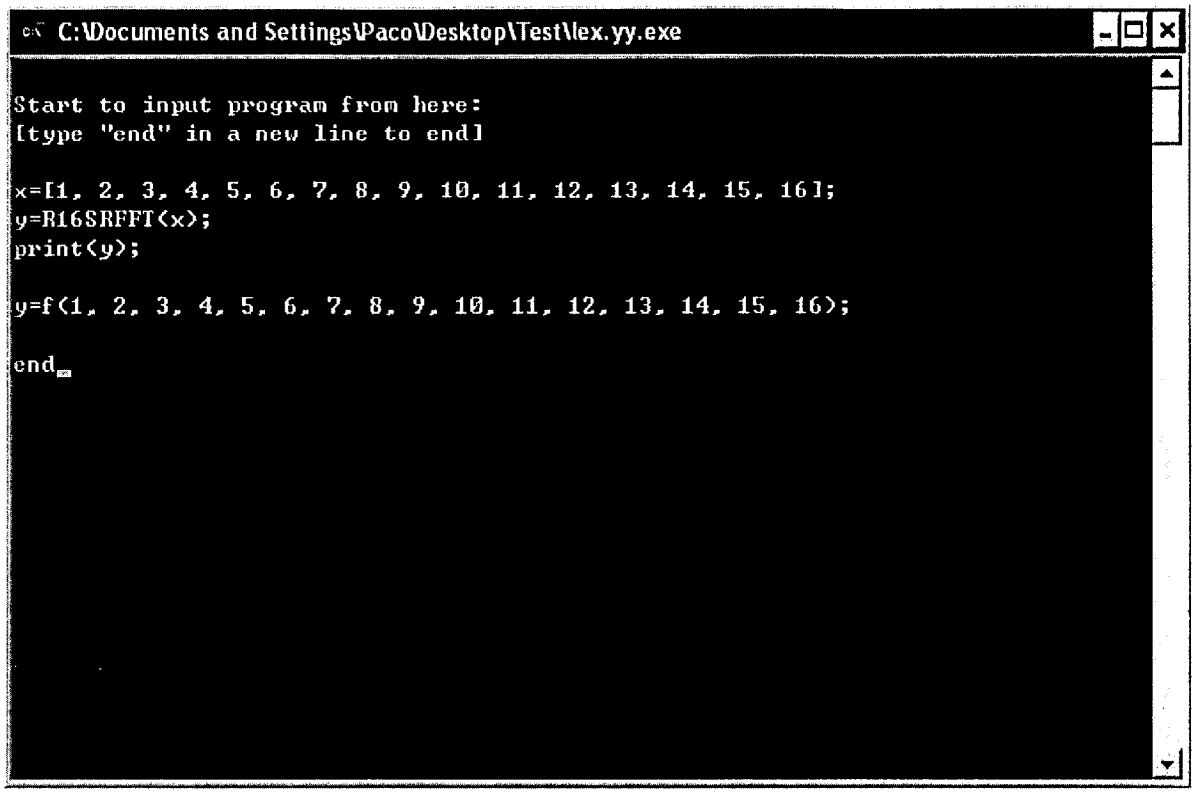



*Figure 6.9 FFT A*

## pf.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "utl.h"

void main(void)
{

Var
*_Temp_66,*_Temp_65,*_Temp_64,*_Temp_63,*_Temp_62,*_Temp_61,*_Temp_60,*_Te
mp_59,*_Temp_58,*_Temp_57,*_Temp_56,*_Temp_55,*_Temp_54,*_Temp_53,*_Temp_5
2,*_Temp_51,*_Temp_50,*_Temp_49,*y,*_Temp_48,*_Temp_47,*_Temp_46,*_Temp_45,
*_Temp_44,*_Temp_43,*_Temp_42,*_Temp_41,*_Temp_40,*_Temp_39,*_Temp_38,*_Te
mp_37,*_Temp_36,*_Temp_35,*_Temp_34,*_Temp_33,*_Temp_32,*_Temp_31,*_Temp_3
0,*_Temp_29,*_Temp_28,*_Temp_27,*_Temp_26,*_Temp_25,*_Temp_24,*_Temp_23,*_T
```

```
emp_22,*_Temp_21,*_Temp_20,*_Temp_19,*_Temp_18,*_Temp_17,*_Temp_16,*_Temp_
15,*_Temp_14,*_Temp_13,*_Temp_12,*_Temp_11,*_Temp_10,*_Temp_9,*_Temp_8,*_Te
mp_7,*_Temp_6,*_Temp_5,*_Temp_4,*_Temp_3,*_Temp_2,*x,*_Temp_1,*pi;

_Temp_66=(Var *)malloc(sizeof(Var)); _Temp_66->tag_=typeNull;
_Temp_65=(Var *)malloc(sizeof(Var)); _Temp_65->tag_=typeNull;
_Temp_64=(Var *)malloc(sizeof(Var)); _Temp_64->tag_=typeNull;
_Temp_63=(Var *)malloc(sizeof(Var)); _Temp_63->tag_=typeNull;
_Temp_62=(Var *)malloc(sizeof(Var)); _Temp_62->tag_=typeNull;
_Temp_61=(Var *)malloc(sizeof(Var)); _Temp_61->tag_=typeNull;
_Temp_60=(Var *)malloc(sizeof(Var)); _Temp_60->tag_=typeNull;
_Temp_59=(Var *)malloc(sizeof(Var)); _Temp_59->tag_=typeNull;
_Temp_58=(Var *)malloc(sizeof(Var)); _Temp_58->tag_=typeNull;
_Temp_57=(Var *)malloc(sizeof(Var)); _Temp_57->tag_=typeNull;
_Temp_56=(Var *)malloc(sizeof(Var)); _Temp_56->tag_=typeNull;
_Temp_55=(Var *)malloc(sizeof(Var)); _Temp_55->tag_=typeNull;
_Temp_54=(Var *)malloc(sizeof(Var)); _Temp_54->tag_=typeNull;
_Temp_53=(Var *)malloc(sizeof(Var)); _Temp_53->tag_=typeNull;
_Temp_52=(Var *)malloc(sizeof(Var)); _Temp_52->tag_=typeNull;
_Temp_51=(Var *)malloc(sizeof(Var)); _Temp_51->tag_=typeNull;
_Temp_50=(Var *)malloc(sizeof(Var)); _Temp_50->tag_=typeNull;
_Temp_49=(Var *)malloc(sizeof(Var)); _Temp_49->tag_=typeNull;
y=(Var *)malloc(sizeof(Var)); y->tag_=typeNull;
_Temp_48=(Var *)malloc(sizeof(Var)); _Temp_48->tag_=typeNull;
_Temp_47=(Var *)malloc(sizeof(Var)); _Temp_47->tag_=typeNull;
_Temp_46=(Var *)malloc(sizeof(Var)); _Temp_46->tag_=typeNull;
_Temp_45=(Var *)malloc(sizeof(Var)); _Temp_45->tag_=typeNull;
_Temp_44=(Var *)malloc(sizeof(Var)); _Temp_44->tag_=typeNull;
_Temp_43=(Var *)malloc(sizeof(Var)); _Temp_43->tag_=typeNull;
_Temp_42=(Var *)malloc(sizeof(Var)); _Temp_42->tag_=typeNull;
_Temp_41=(Var *)malloc(sizeof(Var)); _Temp_41->tag_=typeNull;
_Temp_40=(Var *)malloc(sizeof(Var)); _Temp_40->tag_=typeNull;
_Temp_39=(Var *)malloc(sizeof(Var)); _Temp_39->tag_=typeNull;
_Temp_38=(Var *)malloc(sizeof(Var)); _Temp_38->tag_=typeNull;
_Temp_37=(Var *)malloc(sizeof(Var)); _Temp_37->tag_=typeNull;
_Temp_36=(Var *)malloc(sizeof(Var)); _Temp_36->tag_=typeNull;
_Temp_35=(Var *)malloc(sizeof(Var)); _Temp_35->tag_=typeNull;
_Temp_34=(Var *)malloc(sizeof(Var)); _Temp_34->tag_=typeNull;
_Temp_33=(Var *)malloc(sizeof(Var)); _Temp_33->tag_=typeNull;
_Temp_32=(Var *)malloc(sizeof(Var)); _Temp_32->tag_=typeNull;
_Temp_31=(Var *)malloc(sizeof(Var)); _Temp_31->tag_=typeNull;
_Temp_30=(Var *)malloc(sizeof(Var)); _Temp_30->tag_=typeNull;
_Temp_29=(Var *)malloc(sizeof(Var)); _Temp_29->tag_=typeNull;
_Temp_28=(Var *)malloc(sizeof(Var)); _Temp_28->tag_=typeNull;
_Temp_27=(Var *)malloc(sizeof(Var)); _Temp_27->tag_=typeNull;
_Temp_26=(Var *)malloc(sizeof(Var)); _Temp_26->tag_=typeNull;
_Temp_25=(Var *)malloc(sizeof(Var)); _Temp_25->tag_=typeNull;
_Temp_24=(Var *)malloc(sizeof(Var)); _Temp_24->tag_=typeNull;
```

```
_Temp_23=(Var *)malloc(sizeof(Var)); _Temp_23->tag_=typeNull;
_Temp_22=(Var *)malloc(sizeof(Var)); _Temp_22->tag_=typeNull;
_Temp_21=(Var *)malloc(sizeof(Var)); _Temp_21->tag_=typeNull;
_Temp_20=(Var *)malloc(sizeof(Var)); _Temp_20->tag_=typeNull;
_Temp_19=(Var *)malloc(sizeof(Var)); _Temp_19->tag_=typeNull;
_Temp_18=(Var *)malloc(sizeof(Var)); _Temp_18->tag_=typeNull;
_Temp_17=(Var *)malloc(sizeof(Var)); _Temp_17->tag_=typeNull;
_Temp_16=(Var *)malloc(sizeof(Var)); _Temp_16->tag_=typeNull;
_Temp_15=(Var *)malloc(sizeof(Var)); _Temp_15->tag_=typeNull;
_Temp_14=(Var *)malloc(sizeof(Var)); _Temp_14->tag_=typeNull;
_Temp_13=(Var *)malloc(sizeof(Var)); _Temp_13->tag_=typeNull;
_Temp_12=(Var *)malloc(sizeof(Var)); _Temp_12->tag_=typeNull;
_Temp_11=(Var *)malloc(sizeof(Var)); _Temp_11->tag_=typeNull;
_Temp_10=(Var *)malloc(sizeof(Var)); _Temp_10->tag_=typeNull;
_Temp_9=(Var *)malloc(sizeof(Var)); _Temp_9->tag_=typeNull;
_Temp_8=(Var *)malloc(sizeof(Var)); _Temp_8->tag_=typeNull;
_Temp_7=(Var *)malloc(sizeof(Var)); _Temp_7->tag_=typeNull;
_Temp_6=(Var *)malloc(sizeof(Var)); _Temp_6->tag_=typeNull;
_Temp_5=(Var *)malloc(sizeof(Var)); _Temp_5->tag_=typeNull;
_Temp_4=(Var *)malloc(sizeof(Var)); _Temp_4->tag_=typeNull;
_Temp_3=(Var *)malloc(sizeof(Var)); _Temp_3->tag_=typeNull;
_Temp_2=(Var *)malloc(sizeof(Var)); _Temp_2->tag_=typeNull;
x=(Var *)malloc(sizeof(Var)); x->tag_=typeNull;
_Temp_1=(Var *)malloc(sizeof(Var)); _Temp_1->tag_=typeNull;
pi=(Var *)malloc(sizeof(Var)); pi->tag_=typeNull;



CONST(_Temp_1,3.14159);
SET(pi,_Temp_1);
CONST(_Temp_2,1);
ARR(_Temp_3,_Temp_2);
CONST(_Temp_4,2);
ARR(_Temp_5,_Temp_4);
ARH(_Temp_6,_Temp_3,_Temp_5);
CONST(_Temp_7,3);
ARR(_Temp_8,_Temp_7);
ARH(_Temp_9,_Temp_6,_Temp_8);
CONST(_Temp_10,4);
ARR(_Temp_11,_Temp_10);
ARH(_Temp_12,_Temp_9,_Temp_11);
CONST(_Temp_13,5);
ARR(_Temp_14,_Temp_13);
ARH(_Temp_15,_Temp_12,_Temp_14);
CONST(_Temp_16,6);
ARR(_Temp_17,_Temp_16);
ARH(_Temp_18,_Temp_15,_Temp_17);
CONST(_Temp_19,7);
```

```
ARR(_Temp_20,_Temp_19);
ARH(_Temp_21,_Temp_18,_Temp_20);
CONST(_Temp_22,8);
ARR(_Temp_23,_Temp_22);
ARH(_Temp_24,_Temp_21,_Temp_23);
CONST(_Temp_25,9);
ARR(_Temp_26,_Temp_25);
ARH(_Temp_27,_Temp_24,_Temp_26);
CONST(_Temp_28,10);
ARR(_Temp_29,_Temp_28);
ARH(_Temp_30,_Temp_27,_Temp_29);
CONST(_Temp_31,11);
ARR(_Temp_32,_Temp_31);
ARH(_Temp_33,_Temp_30,_Temp_32);
CONST(_Temp_34,12);
ARR(_Temp_35,_Temp_34);
ARH(_Temp_36,_Temp_33,_Temp_35);
CONST(_Temp_37,13);
ARR(_Temp_38,_Temp_37);
ARH(_Temp_39,_Temp_36,_Temp_38);
CONST(_Temp_40,14);
ARR(_Temp_41,_Temp_40);
ARH(_Temp_42,_Temp_39,_Temp_41);
CONST(_Temp_43,15);
ARR(_Temp_44,_Temp_43);
ARH(_Temp_45,_Temp_42,_Temp_44);
CONST(_Temp_46,16);
ARR(_Temp_47,_Temp_46);
ARH(_Temp_48,_Temp_45,_Temp_47);
SET(x,_Temp_48);
R16SRFFT(_Temp_49,x);
SET(y,_Temp_49);
printvar(_Temp_49);

CONST(_Temp_50,1);
CONST(_Temp_51,2);
CONST(_Temp_52,3);
CONST(_Temp_53,4);
CONST(_Temp_54,5);
CONST(_Temp_55,6);
CONST(_Temp_56,7);
CONST(_Temp_57,8);
CONST(_Temp_58,9);
CONST(_Temp_59,10);
CONST(_Temp_60,11);
CONST(_Temp_61,12);
CONST(_Temp_62,13);
CONST(_Temp_63,14);
```

```
CONST(_Temp_64,15);
CONST(_Temp_65,16);
f(_Temp_66,_Temp_50,_Temp_51,_Temp_52,_Temp_53,_Temp_54,_Temp_55,_Temp_56,_
Temp_57,_Temp_58,_Temp_59,_Temp_60,_Temp_61,_Temp_62,_Temp_63,_Temp_64,_Te
mp_65);
SET(y,_Temp_66);

freevar(_Temp_66);
freevar(_Temp_65);
freevar(_Temp_64);
freevar(_Temp_63);
freevar(_Temp_62);
freevar(_Temp_61);
freevar(_Temp_60);
freevar(_Temp_59);
freevar(_Temp_58);
freevar(_Temp_57);
freevar(_Temp_56);
freevar(_Temp_55);
freevar(_Temp_54);
freevar(_Temp_53);
freevar(_Temp_52);
freevar(_Temp_51);
freevar(_Temp_50);
freevar(_Temp_49);
freevar(y);
freevar(_Temp_48);
freevar(_Temp_47);
freevar(_Temp_46);
freevar(_Temp_45);
freevar(_Temp_44);
freevar(_Temp_43);
freevar(_Temp_42);
freevar(_Temp_41);
freevar(_Temp_40);
freevar(_Temp_39);
freevar(_Temp_38);
freevar(_Temp_37);
freevar(_Temp_36);
freevar(_Temp_35);
freevar(_Temp_34);
freevar(_Temp_33);
freevar(_Temp_32);
freevar(_Temp_31);
freevar(_Temp_30);
freevar(_Temp_29);
freevar(_Temp_28);
freevar(_Temp_27);
```

```
freevar(_Temp_26);
freevar(_Temp_25);
freevar(_Temp_24);
freevar(_Temp_23);
freevar(_Temp_22);
freevar(_Temp_21);
freevar(_Temp_20);
freevar(_Temp_19);
freevar(_Temp_18);
freevar(_Temp_17);
freevar(_Temp_16);
freevar(_Temp_15);
freevar(_Temp_14);
freevar(_Temp_13);
freevar(_Temp_12);
freevar(_Temp_11);
freevar(_Temp_10);
freevar(_Temp_9);
freevar(_Temp_8);
freevar(_Temp_7);
freevar(_Temp_6);
freevar(_Temp_5);
freevar(_Temp_4);
freevar(_Temp_3);
freevar(_Temp_2);
freevar(x);
freevar(_Temp_1);
freevar(pi);


}
```

After we compile pf.c with utl.c and utl.h files, we can get the result:

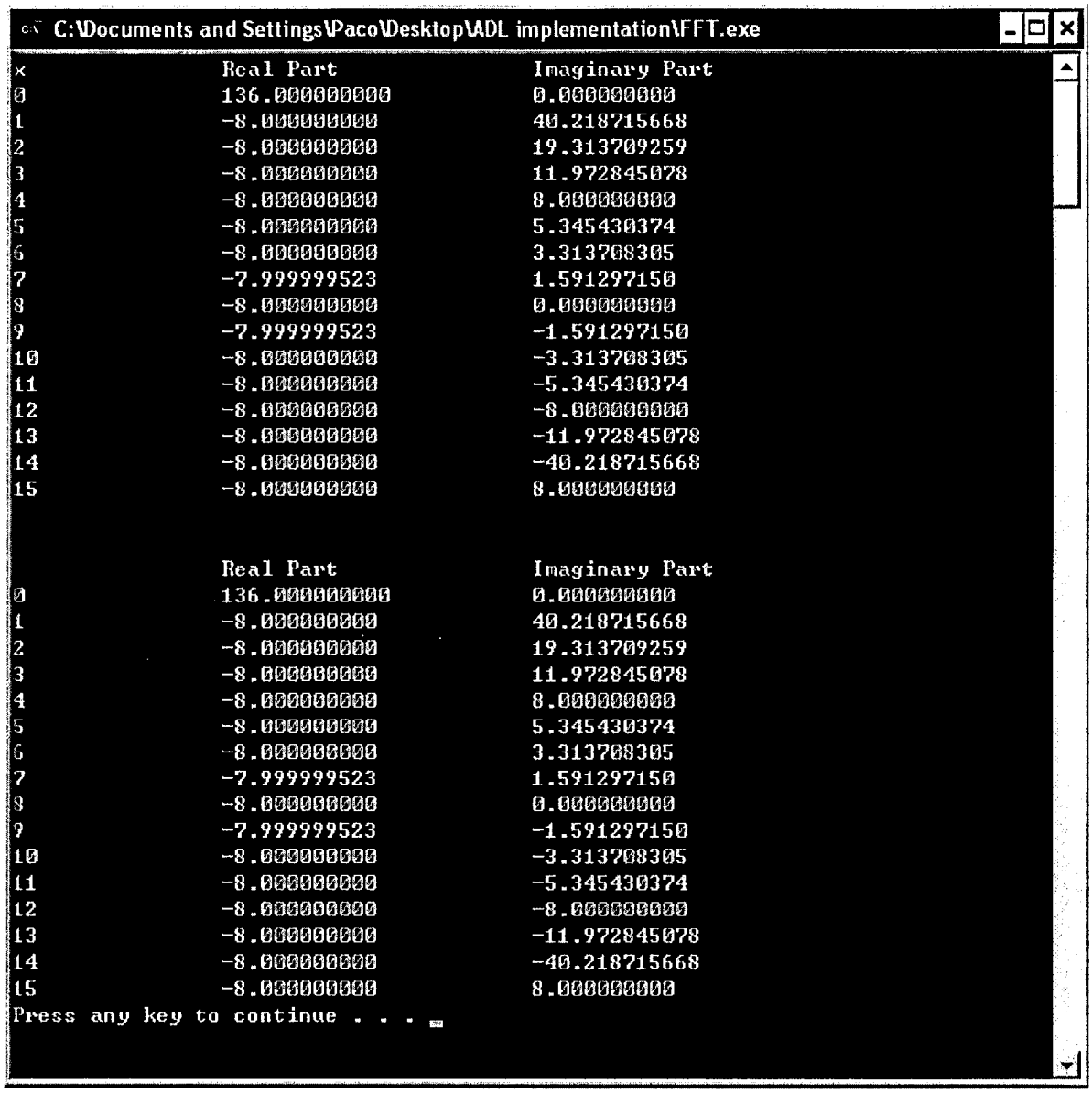



*Figure 6.10 FFT B*

# Chapter Seven

# Conclusions and Future Work

## 7.1 Conclusions

The purpose of this thesis was first to present Application Description Language which is a model-based programming language used for embedded system end-users with a purpose of implementation on Reconfigurable Co-Processor computing system. Also, there is a translator for which to translate the user input ADL to standard ANSI C languages.

PACO v0.1 was developed for the embedded system software design, and system testing purpose. It is non-proprietary and open to all. It addresses the needs of the user, as established by experience with the underlying methods on which it is based. It builds upon similar semantics and notation from C language, Matlab, and other leading methods.

## 7.2 Summary of Contributions

We referred to some previous related work by Rami Abielmona, Mohammad El-Kadri, Mohammad Elbadri, and Nizar Sakr. In their work, they focused on the system architecture design, and hardware system design and implementations. Based on their work, we designed the Application Description Language to the area of Real-Time Reconfigurable Co-Processor Computing as well as the area of Software and Hardware Co-design, and created the corresponding ADL translator to generate C code for Just-In-Time compiler which is introduced by Mohammad El-Kadri. Thus, our approach of translation can work on all

embedded software systems.

## 7.3 Future Research

Although PACO v0.1 defines a precise language, it is not a barrier to future improvements in modeling concepts. We have addressed some leading-edge techniques, such as Models, LEX & YACC, but expect additional techniques to influence future versions of PACO. PACO can be extended without redefining the core translator's coding part.

The extension of PACO v0.1 will be done in Application Description Language part and Translator part. We will not only enlarge the set of languages that can be used in PACO, but also provide user's self-define language functions to let users design their functions by following some syntax rules. That is a concept named open source. To match the development of ADL, the translator will be expanding with more intelligence to translate more complex instructions too.

# References


1. A. V. Aho, R. Sethi, J. D. Ullman. *Compiler: Principles, Techniques, and Tools*, Addison Wesley, 2002.

2. Y. Li, T. Callahan, E. Darnel, R. Harr, U. Kurkure, J. Stockwood, "Hardware Software Co-design of Embedded Reconfigurable Architectures", *Proc. 37th Design Automation Conference*, June 2000.

3. L. Carloni, M. D. Di Benedetto, A. Pinto, A. Sangiovanni-Vincentelli, "Modeling Techniques, Programming Languages, Design Toolsets and Interchange Formats for Hybrid Systems", IST-2001-38314 of European Commission, March 2004.

4. R. Chassaing, *DSP Applications Using C and the TMS320C6x DSK,* Wiley, 2003.

5. Y. Chu, "Introducing A Software Design Language", *Proc. 2nd IEEE International Conference on Software Engineering,* 1976.

6. J. W. Coleman, N. P. Jefferson, C. B. Jones, "Black tie optional: Modeling programming language concepts", Technical Report # CS-TR: 844, School of Computing Science, University of Newcastle upon Tyne, May 2004.

7. CMP, Embedded Systems Programming: *Designing Real-Time System with UML by Bruce Powel Douglass, 2004.* http://www.embedded.com/98/9803fe2.htm

8. R. Corbett. *Bison Manual*, Version 2.0, January 2005.
http://www.gnu.org/software/bison/manual/

9. H. M. Deitel, P. J. Deitel. *C how to program*, Prentice Hall, July 2000.

10. C. Donnelly. *Bison : The Yacc-compatible Parser Generator*, Version 2.1, 2005.
http://www.gnu.org/software/bison/manual/pdf/bison.pdf

11. B. P. Douglass, *UML – The New Language for Real-Time Embedded System (UML 2000)*, 2000. http://wooddes.intranet.gr/papers/Douglass.pdf

12. T. A. Driscoll. *Crash course in MATLAB*. Department of Mathematical Sciences, Ewing Hall, University of Delaware, DE 19716. June 2, 2003.

13. M. El-Kadri, P. Fu, V. Groza, "Software Development Environment for Run-time Reconfigurable System-on-Chip", *Proc. IEEE Intrumentation & Measurement Technology Conference*, May 2005.

14. M. El-Kadri, V. Groza, "Just-in-Time Compiler for a Run-time Reconfigurable Rapid Prototyping Platform", *Proc. IEEE Intrumentation & Measurement Technology Conference*, May 2005.

15. R. Fourer, D. M. Gay, B. W. Kernighan. *AMPL: A Mathematical Programming Language,* Brooks/Cole Publishing Company, 2002.

16. M. Franz, "The Programming Language Lagoona – A Fresh Look at Object-Orientation", *Software: Concepts and Tools*, Springer-Verlag, 1997.

17. E. Gamma, et al. Design Patterns: *Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

18. S. N. Gaulding, J. D. Lawson, "Process Design Engineering – A Methodology for Real-Time Software Development", *Proc. 2nd IEEE International Conference on Software Engineering,* 1976.

19. I. Graham. *Object-Oriented Methods*, Addison-Wesley, 2001

20. V. Groza, R. Abielmona, M. El-Kadri, N. Sakr, M. Elbadri, "A Reconfigurable Co-Processor for Adaptive Embedded Systems", *Proc. of Second Workshop on Intelligent Solutions in Embedded Systems,* May 2004.

21. V. Groza, R. Abielmona, N. Sakr, M. Elbadri, "A Reconfigurable Processor Unit for Embedded Systems *Proc. of Second Workshop on Intelligent Solutions in Embedded Systems,* May 2004.

22. Z. Gu, K. G. Shin, "Synthesis of Real-Time Implementation from UML-RT Models", *Proc. 2nd RTAS Workshop on Model-Driven Embedded Systems*, May 2004.

23. Y-G. Gueheneuc, H. Albin-Amiot, R. Douence, P. Cointe. "Bringing the Gap Between

Modeling and Programming Languages". Technical Report 02/09/INFO, Department of Computer Science, 'Ecole des Mines de Nantes, July 2002

24. B. Hubert. *Lex and Yacc primer/HOWTO*, www.tldp.org/HOWTO/Lex-YACC-HOWTO.html, April 20th, 2002

25. V. Jacobson, "Flex ", version 2.5, March 1995. http://www.gnu.org/software/flex/manual/html_mono/flex.html

26. S. Johnson, "Yet another compiler-compiler", Technical Report CS-32, AT&T Bell Laboratories, Murray Hill, NJ, 1975

27. D. I. Kang, R.Gerber, L. Golubchik, J. K. Hollingsworth, M. Saksena, "A Software Synthesis Tool for Distributed Embedded System Design", *Proc. ACM SIGPLAN workshop on Languages, compilers, and tools for embedded systems*, 1999.

28. B. W. Kernighan, D. M. Ritchie. *The C Programming Language*, Prentice-Hall, N. J., 1978.

29. B. W. Kernighan, D. Ritchie, D. M. Richie. *C Programming Language (2nd Edition)*. Prentice Hall, Inc., 1988.

30. E. Klavins, "A Language for Modeling and Programming Cooperative Control System", *Proc. International Conference on Robotics and Automation*, 2004.

31. C. Kobryn. "Modeling Components and Frameworks with UML," *Communications of the ACM*, vol. 43, no. 10, October 2000.

32. M. Lesk "Lex - A Lexical Analyzer Generator," Comp. Science Technical Report No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

33. J. R. Levine, T. Mason, D. Brown. *Lex & Yacc*, O'Reilly & Associates, 1995.

34. R. C. Martin. *UML Tutorial: Finite State Machines*. Engineering Notebook Column C++ Report, June 1998.

35. National Instruments, *LabVIEW – Getting Started with LabVIEW*, National Instruments Corporate, 11500 North Mopac Exressway Austin, Texas 78759-3504. April 2003.

36. PACT: *The XPP White Paper, Release 2.1 – A Technical Perspective*, 2002.

37. D. Pellerin, S. Thibault. *Practical FPGA Programming in C.* Prentice Hall, 2005.

38. C. Peter, "Hardware Compilation and the Handel-C Language", Oxford University Computing Laboratory, U.K. 1996. http://web.comlab.ox.ac.uk/ouc1/work/christian.peter/overview_handelc.html

39. J. Poskanzer, V. Paxson, *Flex manual*, Free Software Foundation, Cambridge, MA, 1990.

40. E. Schmidt, M.E. Lesk. "Lex - A Lexical Analyzer Generator," Unix Pogrammers's Manual, 7

41. S-B. Scholz, "Single Assignment C – efficient support for high-level array operations in a functional setting", *Journal of Functional Programming*, vol. 13(6), pp. 1005-1059, 2003.

42. L. Shannon, P. Chow. "Using Reconfigurability to Achieve Real-Time Profiling for Hardware/Software Codesign", *Proc. ACM International Symposium on Field Programmable Gate Arrays*, pp. 190-199, 2004.

43. M. Shaw, "Writing Good Software Engineering Research Papers", *Proc. International Conference on Software Engineering,* 2003.

44. B. Sirpatil. *Software Synthesis of SystemC Models*. Masters Thesis, Virginia Polytechnic Institute and State University, July 2002.

45. R. Stallman, Bison: The YACC-compatible Parser Generator, version 1.25, 1995.

46. UML Revision Task Force, *OMG Unified Modeling Language Specification*, v. 1.3, document ad/99-06-08. Object Management Group, June 1999.

# Appedix A paco.l

```
%{
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include "paco.h"
#include "y_tab.h"

#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];


%}

D       [0-9]
S       [ \t]
NL      ((\n)|(\r\n))
SNL     ({S}|{NL})
EL      (\.\.\.)
BS      (\\)
CONT        ({EL}|{BS})
NOT         ((\~)|(\!))
POW         ((\*\*)|(\^))
EPOW        (\.{POW})
IDENT       ([a-zA-Z][_a-zA-Z0-9]*)
EXPON       ([DdEe][+-]?{D}+)
NUMBER      (({D}+\.?{D}*{EXPON}?)|(\.{D}+{EXPON}?)|(0[xX][0-9a-fA-F]+))

%x ML_COMMENT SL_COMMENT INCL USEIT


%%

<<EOF>>                         {
                                    if ( --include_stack_ptr < 0 ){
                                        yyterminate();
                                    }
                                    else{
                                        yy_delete_buffer( YY_CURRENT_BUFFER );
                                        yy_switch_to_buffer( include_stack[include_stack_ptr] );
                                    }
                                }


"/*"                            BEGIN(ML_COMMENT);
<ML_COMMENT>[^*\n]*                 /* eat anything that's not a '*' */
<ML_COMMENT>"*"+[^*/\n]*            /* eat up '*'s not followed by '/'s */
<ML_COMMENT>\n                      ;
<ML_COMMENT>"*"+"/"                 BEGIN(INITIAL);

[/][/][^\n]*[\n]                ;
```

```
"while"                         return WHILE;
"for"                           return FOR;
"print"                         return PRINT;
"do"                            return DO;


{IDENT}                         {
                                    yylval.src = strdup(yytext);
                                    return IDENTIFIER;
                                }

{NUMBER}                        {
                                    yylval.src = strdup(yytext);
                                    return NUMBER;
                                }


">="                            return GE;
"<="                            return LE;
"=="                            return EQ;
"!="                            return NE;

"&&"                            return AND;
"||"                            return OR;
"!"                             return *yytext;

"?"                             return *yytext;

[-()<>=+*/;{}.,:\^\]\[]         {
                                    return *yytext;
                                }

[ \t\r\n]+                      ; /* ignore whitespace */


.                               {
                                    char* sz=malloc(yyleng+20);
                                    strcpy(sz,">>>Unknown token: ");
                                    strcat(sz,yytext);
                                    yyerror(sz);

                                    return LEXERROR;
                                }

%%

yywrap(){return(1);}


int yyerror(char *s) {
    int i;
    unsigned int k;
    char sz[2];sz[1]='\0';

    free(outsrc);
```

```
    outsrc = strdup("");
    AddStr("\nputs(\"!!!Error in source part:\\n\");");

        /*
    AddStr("\nputs(\">>>***** ");
    for(k=0;k<strlen(s);++k){
        sz[0] = s[k];
        if(s[k]=='\"')
            AddStr("\\\"");
        else
            AddStr(sz);
    }
    AddStr(" *****:\");\n");
    */

    i = strlen(yy_current_buffer->yy_ch_buf)-20;
    i = ((i<0)?0:i);
    s = yy_current_buffer->yy_ch_buf+i;

    AddStr("\nputs(\"");
    for(k=0;k<strlen(s);++k){
        sz[0] = s[k];
        if(s[k]=='\"')
            AddStr("\\\"");
        else if(s[k]=='\n' || s[k]=='\r')
            AddStr("\\n");
        else
            AddStr(sz);
    }
    AddStr(" \\n\");\n");

    yyterminate();

    return 0;
}


int __PACOCompile(const char* insrcx, char** outsrcx) {
        char* bodystr;

    YY_BUFFER_STATE buf_state;

        num = 0;
        namestr=NULL;
    include_stack_ptr = 0;

    buf_state = yy_scan_string(insrcx);
    yy_switch_to_buffer(buf_state);

    outsrc = strdup("");
    argstr = strdup("");

    yyparse();

        bodystr = strdup(outsrc);
        free(outsrc);
        outsrc=strdup("");
```

```
    AddStr("#include <stdlib.h>\n");
    AddStr("#include <stdio.h>\n");
    AddStr("#include <string.h>\n");
    AddStr("#include <math.h>\n");
    AddStr("#include \"utl.h\"\n");

    AddStr("\nvoid main(void)\n");
    AddStr("{\n");
    AddStr("\n");

        AddVarDecl();
        AddStr("\n");

        AddStr(bodystr);

        AddStr("\n");
        AddStr("\n");
        ReleaseVar();
    AddStr("\n\n}\n");


    *outsrcx = (char*)malloc((strlen(outsrc) + 1)*sizeof(char));
    strcpy(*outsrcx,outsrc);

    free(argstr);
    free(outsrc);

        yy_delete_buffer(buf_state);
        removeall(&namestr);

        free(bodystr);

    return 0;
}

int __PACOCompileRelease(char** rtntxt) {
    if(*rtntxt!=NULL){
         free(*rtntxt);
         *rtntxt=NULL;
    }

    return 0;
}

void inputstr()
{
        char str[800];

        printf("\nStart to input program from here: \n[type \"end\" in a new line to end]\n\n");
        gets(str);

        while(strcmp(str,"end")!=0){
                in = (char*)realloc(in, (strlen(in) + strlen(str) + 2)*sizeof(char));
                strcat(in, str);

                strcat(in,"\n");
                gets(str);
```

```
        }

        //free(str);
//printf(in);

}

void main( void )
{
    /* yyin = fopen( argv[0], "r" ); */
        //int i;
        //for(i=0;i<10000;++i){
        char *out;
        FILE *fout;


        //in=strdup("y=x=2+8;while(i<10){x=x+1;} x=(s<9);");
        //in=strdup("for(i=0;i<10;i=i+1)for(j=0;j<10;j=j+1){x=x+1;r=0;}");
        //in=strdup("y=x=2+(8+9);x=f(1,3);");
        //in=strdup("x=[1];");
        //in=strdup("x=[1,2,3];");
        //in=strdup("x=[1,2;3,4];d=6;print(d);");
        //in=strdup("x=12;d=6+4-9*2;print(d);");
        //in=strdup("x=[1,2,3;4,5,6];print(x);");
        //in=strdup("                                                    ");

        //in = (char*)strdup(" x=123;");
        //in=malloc(8*sizeof(char));   ////////////////////////////////
        //in[0]=0;

        in=strdup("pi=3.14159;\n                              \n");
        inputstr();

        //printf(in);

if(strlen(in)>0){
        __PACOCompile(in, &out);

        //printf("%s\n",out);

        fout=fopen("pf.c","w");

        fprintf(fout,out);

        fclose(fout);

        __PACOCompileRelease(&out);
}

        //}
        free(in);
        //getchar();

}
```

# Appendix B paco.y

```
%{
#include "paco.h"
%}

%union {
    char* src;
    nodeType *nPtr;             /* node pointer */
};

%expect 0

%token <src> NUMBER
%token <src> IDENTIFIER
%token WHILE FOR   PRINT DO DOWHILE
%token ARRLIST ARGLIST PARA MATRIXLIST
%token EXPRLIST   LEXERROR SEP

%left ',' ';'
%left '='
%left GE LE EQ NE '>' '<'
%left OR
%left AND
%left '+' '-'
%left '*' '/'
%left UMINUS UPLUS
%left NOT
%right '^'
%left '(' '.' '{' '['

%type <nPtr> stmt expr stmt_list expr_list matrix_list

%start program

%%

program         : opt_sep top_stmt opt_sep      { /*exit(0);*/ }
                ;

top_stmt        : top_stmt opt_sep stmt         {
                                                    parserflag = nothing;
                                                    ex($3); freeNode($3);
                                                }
                | stmt                          {
                                                    parserflag = nothing;
                                                    ex($1); freeNode($1);
                                                }
                | LEXERROR                          { YYABORT; }
                ;


stmt            : expr ';'                      { $$ = opr(';', 1, $1); }
                | PRINT expr ';'                { $$ = opr(PRINT, 1, $2); }
                | expr '?'                      { $$ = opr(PRINT, 1, $1); }
                | WHILE '(' expr ')' stmt       { $$ = opr(WHILE, 2, $3, $5); }
```

```
                | DO '{' stmt_list '}' WHILE '(' expr ')' ';'
                                                { $$ = opr(DOWHILE, 2, $3,
$7); }
                | FOR '(' expr ';' expr ';' expr ')' stmt
                                                { $$ = opr(FOR, 4, $3, $5, $7,
$9); }
                | '{' stmt_list '}'             { $$ = $2; }
                ;

expr            : NUMBER                        { $$ = con($1); }
                | IDENTIFIER                    { $$ = id($1); }
                | '+' expr %prec UPLUS          { $$ = $2; }
                | '-' expr %prec UMINUS         { $$ = opr(UMINUS, 1, $2); }
                | '!' expr %prec NOT            { $$ = opr(NOT, 1, $2); }
                | expr '+' expr                 { $$ = opr('+', 2, $1, $3); }
                | expr '-' expr                 { $$ = opr('-', 2, $1, $3); }
                | expr '*' expr                 { $$ = opr('*', 2, $1, $3); }
                | expr '/' expr                 { $$ = opr('/', 2, $1, $3); }
                | expr '<' expr                 { $$ = opr('<', 2, $1, $3); }
                | expr '>' expr                 { $$ = opr('>', 2, $1, $3); }
                | expr '^' expr                 { $$ = opr('^', 2, $1, $3); }
                | expr AND expr                 { $$ = opr(AND, 2, $1, $3); }
                | expr OR expr                  { $$ = opr(OR, 2, $1, $3); }
                | expr GE expr                  { $$ = opr(GE, 2, $1, $3); }
                | expr LE expr                  { $$ = opr(LE, 2, $1, $3); }
                | expr NE expr                  { $$ = opr(NE, 2, $1, $3); }
                | expr EQ expr                  { $$ = opr(EQ, 2, $1, $3); }
                | '(' expr ')'                  { $$ = $2; }
                | IDENTIFIER '(' expr_list ')'  { $$ = opr(PARA, 2, id($1), $3); }
                | IDENTIFIER '(' ')'            { $$ = opr(PARA, 1, id($1)); }
                        | '[' ']'                       { $$ = opr(ARRLIST, 0); }
                | '[' matrix_list ']'           { $$ = opr(ARRLIST, 1, $2); }
                | IDENTIFIER '=' expr           { $$ = opr('=', 2, id($1), $3); }
                ;

expr_list       : expr                          { $$ = opr(EXPRLIST, 1, $1); }
                | expr_list ',' expr            { $$ = opr(EXPRLIST, 2, $3, $1); }
                ;

matrix_list     : expr_list                     { $$ = opr(MATRIXLIST, 1, $1); }
                | matrix_list ';' expr_list     { $$ = opr(MATRIXLIST, 2, $3, $1); }
                        ;

stmt_list       : stmt opt_sep                  { $$ = $1; }
                | stmt_list stmt opt_sep        { $$ = opr(';', 2, $1, $2); }
                ;

 sep            : ','
                | ';'
                | '\n'
                | sep ','
                | sep ';'
                | sep '\n'
                ;
```

```
opt_sep             : // empty
                      | sep
                      ;


%%
```

# Appendix C lex.yy.c

```
/* A lexical scanner generated by flex */


#define FLEX_SCANNER
#define YY_FLEX_MAJOR_VERSION 2
#define YY_FLEX_MINOR_VERSION 5


#include <stdio.h>


/* cfront 1.2 defines "c_plusplus" instead of "__cplusplus" */
#ifdef c_plusplus
#ifndef __cplusplus
#define __cplusplus
#endif
#endif


#ifdef __cplusplus

#include <stdlib.h>
#include <unistd.h>

/* Use prototypes in function declarations. */
#define YY_USE_PROTOS

/* The "const" storage-class-modifier is valid. */
#define YY_USE_CONST

#else   /* ! __cplusplus */

#if __STDC__

#define YY_USE_PROTOS
#define YY_USE_CONST

#endif  /* __STDC__ */
#endif  /* ! __cplusplus */

#ifdef __TURBOC__
 #pragma warn -rch
 #pragma warn -use
#include <io.h>
#include <stdlib.h>
#define YY_USE_CONST
#define YY_USE_PROTOS
#endif

#ifdef YY_USE_CONST
#define yyconst const
#else
#define yyconst
#endif
```

```
#ifdef YY_USE_PROTOS
#define YY_PROTO(proto) proto
#else
#define YY_PROTO(proto) ()
#endif

/* Returned upon end-of-file. */
#define YY_NULL 0

/* Promotes a possibly negative, possibly signed char to an unsigned
 * integer for use as an array index.  If the signed char is negative,
 * we want to instead treat it as an 8-bit unsigned char, hence the
 * double cast.
 */
#define YY_SC_TO_UI(c) ((unsigned int) (unsigned char) c)

/* Enter a start condition.  This macro really ought to take a parameter,
 * but we do it the disgusting crufty way forced on us by the ()-less
 * definition of BEGIN.
 */
#define BEGIN yy_start = 1 + 2 *

/* Translate the current start state into a value that can be later handed
 * to BEGIN to return to the state.  The YYSTATE alias is for lex
 * compatibility.
 */
#define YY_START ((yy_start - 1) / 2)
#define YYSTATE YY_START

/* Action number for EOF rule of a given start state. */
#define YY_STATE_EOF(state) (YY_END_OF_BUFFER + state + 1)

/* Special action meaning "start processing a new file". */
#define YY_NEW_FILE yyrestart( yyin )

#define YY_END_OF_BUFFER_CHAR 0

/* Size of default input buffer. */
#define YY_BUF_SIZE 16384

typedef struct yy_buffer_state *YY_BUFFER_STATE;

extern int yyleng;
extern FILE *yyin, *yyout;

#define EOB_ACT_CONTINUE_SCAN 0
#define EOB_ACT_END_OF_FILE 1
#define EOB_ACT_LAST_MATCH 2

/* The funky do-while in the following #define is used to turn the definition
 * int a single C statement (which needs a semi-colon terminator).  This
 * avoids problems with code like:
 *
 *      if ( condition_holds )
 *              yyless( 5 );
 *      else
```

```
 *                      do_something_else();
 *
 * Prior to using the do-while the compiler would get upset at the
 * "else" because it interpreted the "if" statement as being all
 * done when it reached the ';' after the yyless() call.
 */

/* Return all but the first 'n' matched characters back to the input stream. */

#define yyless(n) \
        do \
                { \
                /* Undo effects of setting up yytext. */ \
                *yy_cp = yy_hold_char; \
                yy_c_buf_p = yy_cp = yy_bp + n - YY_MORE_ADJ; \
                YY_DO_BEFORE_ACTION; /* set up yytext again */ \
                } \
        while ( 0 )

#define unput(c) yyunput( c, yytext_ptr )

/* The following is because we cannot portably get our hands on size_t
 * (without autoconf's help, which isn't available because we want
 * flex-generated scanners to compile on their own).
 */
typedef unsigned int yy_size_t;


struct yy_buffer_state
        {
        FILE *yy_input_file;

        char *yy_ch_buf;            /* input buffer */
        char *yy_buf_pos;                   /* current position in input buffer */

        /* Size of input buffer in bytes, not including room for EOB
         * characters.
         */
        yy_size_t yy_buf_size;

        /* Number of characters read into yy_ch_buf, not including EOB
         * characters.
         */
        int yy_n_chars;

        /* Whether we "own" the buffer - i.e., we know we created it,
         * and can realloc() it to grow it, and should free() it to
         * delete it.
         */
        int yy_is_our_buffer;

        /* Whether this is an "interactive" input source; if so, and
         * if we're using stdio for input, then we want to use getc()
         * instead of fread(), to make sure we stop fetching input after
         * each newline.
         */
        int yy_is_interactive;
```

```
        /* Whether we're considered to be at the beginning of a line.
         * If so, '^' rules will be active on the next match, otherwise
         * not.
         */
        int yy_at_bol;

        /* Whether to try to fill the input buffer when we reach the
         * end of it.
         */
        int yy_fill_buffer;

        int yy_buffer_status;
#define YY_BUFFER_NEW 0
#define YY_BUFFER_NORMAL 1
        /* When an EOF's been seen but there's still some text to process
         * then we mark the buffer as YY_EOF_PENDING, to indicate that we
         * shouldn't try reading from the input source any more.  We might
         * still have a bunch of tokens to match, though, because of
         * possible backing-up.
         *
         * When we actually see the EOF, we change the status to "new"
         * (via yyrestart()), so that the user can continue scanning by
         * just pointing yyin at a new input file.
         */
#define YY_BUFFER_EOF_PENDING 2
        };

static YY_BUFFER_STATE yy_current_buffer = 0;

/* We provide macros for accessing buffer states in case in the
 * future we want to put the buffer states in a more general
 * "scanner state".
 */
#define YY_CURRENT_BUFFER yy_current_buffer


/* yy_hold_char holds the character lost when yytext is formed. */
static char yy_hold_char;

static int yy_n_chars;          /* number of characters read into yy_ch_buf */


int yyleng;

/* Points to current character in buffer. */
static char *yy_c_buf_p = (char *) 0;
static int yy_init = 1;         /* whether we need to initialize */
static int yy_start = 0;        /* start state number */

/* Flag which is used to allow yywrap()'s to do buffer switches
 * instead of setting up a fresh yyin.  A bit of a hack ...
 */
static int yy_did_buffer_switch_on_eof;

void yyrestart YY_PROTO(( FILE *input_file ));

void yy_switch_to_buffer YY_PROTO(( YY_BUFFER_STATE new_buffer ));
void yy_load_buffer_state YY_PROTO(( void ));
```

```
YY_BUFFER_STATE yy_create_buffer YY_PROTO(( FILE *file, int size ));
void yy_delete_buffer YY_PROTO(( YY_BUFFER_STATE b ));
void yy_init_buffer YY_PROTO(( YY_BUFFER_STATE b, FILE *file ));
void yy_flush_buffer YY_PROTO(( YY_BUFFER_STATE b ));
#define YY_FLUSH_BUFFER yy_flush_buffer( yy_current_buffer )

YY_BUFFER_STATE yy_scan_buffer YY_PROTO(( char *base, yy_size_t size ));
YY_BUFFER_STATE yy_scan_string YY_PROTO(( yyconst char *str ));
YY_BUFFER_STATE yy_scan_bytes YY_PROTO(( yyconst char *bytes, int len ));

static void *yy_flex_alloc YY_PROTO(( yy_size_t ));
static void *yy_flex_realloc YY_PROTO(( void *, yy_size_t ));
static void yy_flex_free YY_PROTO(( void * ));

#define yy_new_buffer yy_create_buffer

#define yy_set_interactive(is_interactive) \
        { \
        if ( ! yy_current_buffer ) \
                yy_current_buffer = yy_create_buffer( yyin, YY_BUF_SIZE ); \
        yy_current_buffer->yy_is_interactive = is_interactive; \
        }

#define yy_set_bol(at_bol) \
        { \
        if ( ! yy_current_buffer ) \

                yy_current_buffer = yy_create_buffer( yyin, YY_BUF_SIZE ); \
        yy_current_buffer->yy_at_bol = at_bol; \
        }

#define YY_AT_BOL() (yy_current_buffer->yy_at_bol)

typedef unsigned char YY_CHAR;
FILE *yyin = (FILE *) 0, *yyout = (FILE *) 0;
typedef int yy_state_type;
extern char *yytext;
#define yytext_ptr yytext

static yy_state_type yy_get_previous_state YY_PROTO(( void ));
static yy_state_type yy_try_NUL_trans YY_PROTO(( yy_state_type current_state ));
static int yy_get_next_buffer YY_PROTO(( void ));
static void yy_fatal_error YY_PROTO(( yyconst char msg[] ));

/* Done after the current pattern has been matched and before the
 * corresponding action - sets up yytext.
 */
#define YY_DO_BEFORE_ACTION \
        yytext_ptr = yy_bp; \
        yyleng = (int) (yy_cp - yy_bp); \
        yy_hold_char = *yy_cp; \
        *yy_cp = '\0'; \
        yy_c_buf_p = yy_cp;

#define YY_NUM_RULES 24
#define YY_END_OF_BUFFER 25
static yyconst short int yy_accept[76] =
    {   0,
```

```
        0,    0,    2,    2,    0,    0,    0,    0,    0,    0,
       25,   23,   22,   22,   19,   23,   21,   21,   21,   12,
       12,   21,   21,   21,   20,   11,   11,   11,   11,   11,
       23,    2,    4,    3,   24,   22,   16,   17,   12,    1,
        0,   12,   12,    0,    0,   14,   15,   13,   11,   10,
       11,   11,   11,   18,    2,    3,    3,    5,    0,    0,
        6,   12,    0,   12,   12,    8,   11,   11,    0,   12,
       11,   11,    9,    7,    0
    } ;

static yyconst int yy_ec[256] =
    {   0,
        1,    1,    1,    1,    1,    1,    1,    1,    2,    3,
        1,    1,    2,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    2,    4,    1,    1,    1,    1,    5,    1,    6,
        6,    7,    8,    6,    8,    9,   10,   11,   12,   12,
       12,   12,   12,   12,   12,   12,   12,    6,    6,   13,
       14,   15,   16,    1,   17,   17,   17,   18,   18,   17,
       19,   19,   19,   19,   19,   19,   19,   19,   19,   19,
       19,   19,   19,   19,   19,   19,   19,   20,   19,   19,
        6,    1,    6,    6,   21,    1,   17,   17,   17,   22,

       23,   24,   19,   25,   26,   19,   19,   27,   19,   28,
       29,   30,   19,   31,   19,   32,   19,   19,   33,   20,
       19,   19,    6,   34,    6,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,

        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1
    } ;

static yyconst int yy_meta[35] =
    {   0,
        1,    1,    2,    1,    1,    1,    3,    1,    1,    1,
        4,    4,    1,    1,    1,    1,    4,    4,    5,    5,
        5,    4,    4,    4,    5,    5,    5,    5,    5,    5,
        5,    5,    5,    1
    } ;

static yyconst short int yy_base[83] =
    {   0,
        0,    0,   32,   33,    0,    0,    0,    0,    0,    0,
      137,  138,   35,   39,  122,  130,  138,   32,   38,   40,
      114,  119,  118,  117,  138,    0,  101,  100,   97,   94,
       81,    0,  138,   40,  138,   51,  138,  138,   53,  138,
      111,   55,   76,   71,    0,  138,  138,  138,    0,    0,
       61,   54,   48,  138,    0,   49,   62,  138,   73,   67,
```

```
    138,   75,   77,   79,    0,    0,   40,   34,   83,   88,
     25,   32,    0,    0,  138,  100,  105,  107,  112,  117,
    122,   42
    } ;

static yyconst short int yy_def[83] =
    {   0,
     75,    1,   76,   76,   77,   77,   77,   77,   77,   77,
     75,   75,   75,   75,   75,   75,   75,   75,   75,   75,
     20,   75,   75,   75,   75,   78,   78,   78,   78,   78,
     75,   79,   75,   80,   75,   75,   75,   75,   75,   75,
     81,   75,   20,   75,   82,   75,   75,   75,   78,   78,
     78,   78,   78,   75,   79,   80,   80,   75,   75,   81,
     75,   75,   75,   75,   82,   78,   78,   78,   75,   75,
     78,   78,   78,   78,    0,   75,   75,   75,   75,   75,
     75,   75
    } ;

static yyconst short int yy_nxt[173] =
    {   0,
     12,   13,   14,   15,   16,   17,   17,   17,   18,   19,
     20,   21,   22,   23,   24,   25,   26,   26,   26,   26,
     12,   27,   26,   28,   26,   26,   26,   26,   26,   29,
     26,   26,   30,   31,   33,   33,   36,   36,   34,   34,
     36,   36,   39,   39,   40,   65,   57,   41,   42,   58,
     43,   43,   36,   36,   74,   75,   73,   44,   75,   45,
     72,   44,   44,   39,   39,   62,   62,   71,   57,   61,
     59,   58,   44,   68,   59,   59,   44,   44,   63,   67,
     69,   64,   64,   70,   70,   62,   62,   64,   64,   64,
     64,   66,   44,   70,   70,   75,   44,   44,   70,   70,

     32,   32,   32,   32,   32,   35,   35,   35,   35,   35,
     49,   49,   55,   61,   54,   55,   55,   56,   53,   56,
     56,   56,   60,   60,   60,   60,   60,   52,   51,   50,
     48,   47,   46,   75,   38,   37,   75,   11,   75,   75,
     75,   75,   75,   75,   75,   75,   75,   75,   75,   75,
     75,   75,   75,   75,   75,   75,   75,   75,   75,   75,
     75,   75
    } ;

static yyconst short int yy_chk[173] =
    {   0,
      1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
      1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
      1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
      1,    1,    1,    1,    3,    4,   13,   13,    3,    4,
     14,   14,   18,   18,   19,   82,   34,   19,   20,   34,
     20,   20,   36,   36,   72,   56,   71,   20,   56,   20,
     68,   20,   20,   39,   39,   42,   42,   67,   57,   60,
     39,   57,   42,   53,   39,   39,   42,   42,   44,   52,
     59,   44,   44,   59,   59,   62,   62,   63,   63,   64,
     64,   51,   62,   69,   69,   43,   62,   62,   70,   70,

     76,   76,   76,   76,   76,   77,   77,   77,   77,   77,
     78,   78,   79,   41,   31,   79,   79,   80,   30,   80,
     80,   80,   81,   81,   81,   81,   81,   29,   28,   27,
     24,   23,   22,   21,   16,   15,   11,   75,   75,   75,
```

```
        75,   75,   75,   75,   75,   75,   75,   75,   75,   75,
        75,   75,   75,   75,   75,   75,   75,   75,   75,   75,
        75,   75,   75,   75,   75,   75,   75,   75,   75,   75,
        75,   75
    } ;

static yy_state_type yy_last_accepting_state;
static char *yy_last_accepting_cpos;

/* The intent behind this definition is that it'll catch
 * any uses of REJECT which flex missed.
 */
#define REJECT reject_used_but_not_detected
#define yymore() yymore_used_but_not_detected
#define YY_MORE_ADJ 0
char *yytext;
#line 1 "paco.l"
#define INITIAL 0
#line 2 "paco.l"
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include "paco.h"
#include "y_tab.h"

#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];


#define ML_COMMENT 1
#define SL_COMMENT 2
#define INCL 3
#define USEIT 4

#line 439 "lex.yy.c"

/* Macros after this point can all be overridden by user definitions in
 * section 1.
 */

#ifndef YY_SKIP_YYWRAP
#ifdef __cplusplus
extern "C" int yywrap YY_PROTO(( void ));
#else
extern int yywrap YY_PROTO(( void ));
#endif
#endif

#ifndef YY_NO_UNPUT
static void yyunput YY_PROTO(( int c, char *buf_ptr ));
#endif

#ifndef yytext_ptr
static void yy_flex_strncpy YY_PROTO(( char *, yyconst char *, int ));
#endif

#ifndef YY_NO_INPUT
#ifdef __cplusplus
```

```
static int yyinput YY_PROTO(( void ));
#else
static int input YY_PROTO(( void ));
#endif
#endif

#if YY_STACK_USED
static int yy_start_stack_ptr = 0;
static int yy_start_stack_depth = 0;
static int *yy_start_stack = 0;
#ifndef YY_NO_PUSH_STATE
static void yy_push_state YY_PROTO(( int new_state ));
#endif
#ifndef YY_NO_POP_STATE
static void yy_pop_state YY_PROTO(( void ));
#endif
#ifndef YY_NO_TOP_STATE
static int yy_top_state YY_PROTO(( void ));
#endif

#else
#define YY_NO_PUSH_STATE 1
#define YY_NO_POP_STATE 1
#define YY_NO_TOP_STATE 1
#endif

#ifdef YY_MALLOC_DECL
YY_MALLOC_DECL
#else
#if __STDC__
#ifndef __cplusplus
#include <stdlib.h>
#endif
#else
/* Just try to get by without declaring the routines.  This will fail
 * miserably on non-ANSI systems for which sizeof(size_t) != sizeof(int)
 * or sizeof(void*) != sizeof(int).
 */
#endif
#endif

/* Amount of stuff to slurp up with each read. */
#ifndef YY_READ_BUF_SIZE
#define YY_READ_BUF_SIZE 8192
#endif

/* Copy whatever the last rule matched to the standard output. */

#ifndef ECHO
/* This used to be an fputs(), but since the string might contain NUL's,
 * we now use fwrite().
 */
#define ECHO (void) fwrite( yytext, yyleng, 1, yyout )
#endif

/* Gets input and stuffs it into "buf".  number of characters read, or YY_NULL,
 * is returned in "result".
 */
```

```
#ifndef YY_INPUT
#define YY_INPUT(buf,result,max_size) \
        if ( yy_current_buffer->yy_is_interactive ) \
                { \
                int c = '*', n; \
                for ( n = 0; n < max_size && \
                             (c = getc( yyin )) != EOF && c != '\n'; ++n ) \
                        buf[n] = (char) c; \
                if ( c == '\n' ) \
                        buf[n++] = (char) c; \
                if ( c == EOF && ferror( yyin ) ) \
                        YY_FATAL_ERROR( "input in flex scanner failed" ); \
                result = n; \
                } \
        else if ( ((result = fread( buf, 1, max_size, yyin )) == 0) \
                  && ferror( yyin ) ) \
                YY_FATAL_ERROR( "input in flex scanner failed" );
#endif

/* No semi-colon after return; correct usage is to write "yyterminate();" -
 * we don't want an extra ';' after the "return" because that will cause
 * some compilers to complain about unreachable statements.
 */
#ifndef yyterminate
#define yyterminate() return YY_NULL
#endif

/* Number of entries by which start-condition stack grows. */
#ifndef YY_START_STACK_INCR
#define YY_START_STACK_INCR 25
#endif

/* Report a fatal error. */
#ifndef YY_FATAL_ERROR
#define YY_FATAL_ERROR(msg) yy_fatal_error( msg )
#endif

/* Default declaration of generated scanner - a define so the user can
 * easily add parameters.
 */
#ifndef YY_DECL
#define YY_DECL int yylex YY_PROTO(( void ))
#endif

/* Code executed at the beginning of each rule, after yytext and yyleng
 * have been set up.
 */
#ifndef YY_USER_ACTION
#define YY_USER_ACTION
#endif

/* Code executed at the end of each rule. */
#ifndef YY_BREAK
#define YY_BREAK break;
#endif

#define YY_RULE_SETUP \
        YY_USER_ACTION
```

```
YY_DECL
        {
        register yy_state_type yy_current_state;
        register char *yy_cp, *yy_bp;
        register int yy_act;

#line 31 "paco.l"


#line 589 "lex.yy.c"

        if ( yy_init )
                {
                yy_init = 0;

#ifdef YY_USER_INIT
                YY_USER_INIT;
#endif

                if ( ! yy_start )
                        yy_start = 1;       /* first start state */

                if ( ! yyin )
                        yyin = stdin;

                if ( ! yyout )
                        yyout = stdout;

                if ( ! yy_current_buffer )
                        yy_current_buffer =
                                yy_create_buffer( yyin, YY_BUF_SIZE );

                yy_load_buffer_state();
                }

        while ( 1 )                     /* loops until end-of-file is reached */
                {
                yy_cp = yy_c_buf_p;

                /* Support of yytext. */
                *yy_cp = yy_hold_char;

                /* yy_bp points to the position in yy_ch_buf of the start of
                 * the current run.
                 */
                yy_bp = yy_cp;

                yy_current_state = yy_start;
yy_match:
                do
                        {
                        register YY_CHAR yy_c = yy_ec[YY_SC_TO_UI(*yy_cp)];
                        if ( yy_accept[yy_current_state] )
                                {
                                yy_last_accepting_state = yy_current_state;
                                yy_last_accepting_cpos = yy_cp;
                                }
```

```
                        while ( yy_chk[yy_base[yy_current_state] + yy_c] != yy_current_state )
                                {
                                yy_current_state = (int) yy_def[yy_current_state];
                                if ( yy_current_state >= 76 )
                                        yy_c = yy_meta[(unsigned int) yy_c];
                                }
                        yy_current_state = yy_nxt[yy_base[yy_current_state] + (unsigned int) yy_c];
                        ++yy_cp;
                        }
                while ( yy_base[yy_current_state] != 138 );

yy_find_action:
                yy_act = yy_accept[yy_current_state];
                if ( yy_act == 0 )
                        { /* have to back up */
                        yy_cp = yy_last_accepting_cpos;
                        yy_current_state = yy_last_accepting_state;
                        yy_act = yy_accept[yy_current_state];
                        }

                YY_DO_BEFORE_ACTION;


do_action:      /* This label is used only to access EOF actions. */


                switch ( yy_act )
        { /* beginning of action switch */
                        case 0: /* must back up */
                        /* undo the effects of YY_DO_BEFORE_ACTION */
                        *yy_cp = yy_hold_char;
                        yy_cp = yy_last_accepting_cpos;
                        yy_current_state = yy_last_accepting_state;
                        goto yy_find_action;

case YY_STATE_EOF(INITIAL):
case YY_STATE_EOF(ML_COMMENT):
case YY_STATE_EOF(SL_COMMENT):
case YY_STATE_EOF(INCL):
case YY_STATE_EOF(USEIT):
#line 33 "paco.l"
{
                                        if ( --include_stack_ptr < 0 ){
                                            yyterminate();
                                        }
                                        else{
                                            yy_delete_buffer( YY_CURRENT_BUFFER );
                                            yy_switch_to_buffer( include_stack[include_stack_ptr] );
                                        }
                                    }
        YY_BREAK
case 1:
YY_RULE_SETUP
#line 44 "paco.l"
BEGIN(ML_COMMENT);
        YY_BREAK
case 2:
YY_RULE_SETUP
```

```
#line 45 "paco.l"
/* eat anything that's not a '*' */
	YY_BREAK
case 3:
YY_RULE_SETUP
#line 46 "paco.l"
/* eat up '*'s not followed by '/'s */
	YY_BREAK
case 4:
YY_RULE_SETUP
#line 47 "paco.l"
;
	YY_BREAK
case 5:
YY_RULE_SETUP
#line 48 "paco.l"
BEGIN(INITIAL);
	YY_BREAK
case 6:
YY_RULE_SETUP
#line 50 "paco.l"
;
	YY_BREAK
case 7:
YY_RULE_SETUP
#line 52 "paco.l"
return WHILE;
	YY_BREAK
case 8:
YY_RULE_SETUP
#line 53 "paco.l"
return FOR;
	YY_BREAK
case 9:
YY_RULE_SETUP
#line 54 "paco.l"
return PRINT;
	YY_BREAK
case 10:
YY_RULE_SETUP
#line 55 "paco.l"
return DO;
	YY_BREAK
case 11:
YY_RULE_SETUP
#line 59 "paco.l"
{
                                        yylval.src = strdup(yytext);
                                        return IDENTIFIER;
                                    }
	YY_BREAK
case 12:
YY_RULE_SETUP
#line 64 "paco.l"
{
                                        yylval.src = strdup(yytext);
                                        return NUMBER;
                                    }
```

```
        YY_BREAK
case 13:
YY_RULE_SETUP
#line 70 "paco.l"
return GE;
        YY_BREAK
case 14:
YY_RULE_SETUP
#line 71 "paco.l"
return LE;
        YY_BREAK
case 15:
YY_RULE_SETUP
#line 72 "paco.l"
return EQ;
        YY_BREAK
case 16:
YY_RULE_SETUP
#line 73 "paco.l"
return NE;
        YY_BREAK
case 17:
YY_RULE_SETUP
#line 75 "paco.l"
return AND;
        YY_BREAK
case 18:
YY_RULE_SETUP
#line 76 "paco.l"
return OR;
        YY_BREAK
case 19:
YY_RULE_SETUP
#line 77 "paco.l"
return *yytext;
        YY_BREAK
case 20:
YY_RULE_SETUP
#line 79 "paco.l"
return *yytext;
        YY_BREAK
case 21:
YY_RULE_SETUP
#line 81 "paco.l"
{
                                        return *yytext;
                                }
        YY_BREAK
case 22:
YY_RULE_SETUP
#line 85 "paco.l"
;   /* ignore whitespace */
        YY_BREAK
case 23:
YY_RULE_SETUP
#line 88 "paco.l"
{
                                        char* sz=malloc(yyleng+20);
```

```
                                        strcpy(sz,">>>Unknown token: ");
                                        strcat(sz,yytext);
                                        yyerror(sz);

                                        return LEXERROR;
                                    }
        YY_BREAK
case 24:
YY_RULE_SETUP
#line 97 "paco.l"
ECHO;
        YY_BREAK
#line 823 "lex.yy.c"

        case YY_END_OF_BUFFER:
                {
                /* Amount of text matched not including the EOB char. */
                int yy_amount_of_matched_text = (int) (yy_cp - yytext_ptr) - 1;

                /* Undo the effects of YY_DO_BEFORE_ACTION. */
                *yy_cp = yy_hold_char;

                if ( yy_current_buffer->yy_buffer_status == YY_BUFFER_NEW )
                        {
                        /* We're scanning a new file or input source.  It's
                         * possible that this happened because the user
                         * just pointed yyin at a new source and called
                         * yylex().  If so, then we have to assure
                         * consistency between yy_current_buffer and our
                         * globals.  Here is the right place to do so, because
                         * this is the first action (other than possibly a
                         * back-up) that will match for the new input source.
                         */
                        yy_n_chars = yy_current_buffer->yy_n_chars;
                        yy_current_buffer->yy_input_file = yyin;
                        yy_current_buffer->yy_buffer_status = YY_BUFFER_NORMAL;
                        }

                /* Note that here we test for yy_c_buf_p "<=" to the position
                 * of the first EOB in the buffer, since yy_c_buf_p will
                 * already have been incremented past the NUL character
                 * (since all states make transitions on EOB to the
                 * end-of-buffer state).  Contrast this with the test
                 * in input().
                 */
                if ( yy_c_buf_p <= &yy_current_buffer->yy_ch_buf[yy_n_chars] )
                        { /* This was really a NUL. */
                        yy_state_type yy_next_state;

                        yy_c_buf_p = yytext_ptr + yy_amount_of_matched_text;

                        yy_current_state = yy_get_previous_state();

                        /* Okay, we're now positioned to make the NUL
                         * transition.  We couldn't have
                         * yy_get_previous_state() go ahead and do it
                         * for us because it doesn't know how to deal
                         * with the possibility of jamming (and we don't
```

```
                * want to build jamming into it because then it
                * will run more slowly).
                */

               yy_next_state = yy_try_NUL_trans( yy_current_state );

               yy_bp = yytext_ptr + YY_MORE_ADJ;

               if ( yy_next_state )
                       {
                       /* Consume the NUL. */
                       yy_cp = ++yy_c_buf_p;
                       yy_current_state = yy_next_state;
                       goto yy_match;
                       }

               else
                       {
                       yy_cp = yy_c_buf_p;
                       goto yy_find_action;
                       }
               }

       else switch ( yy_get_next_buffer() )
               {
               case EOB_ACT_END_OF_FILE:
                       {
                       yy_did_buffer_switch_on_eof = 0;

                       if ( yywrap() )
                               {
                               /* Note: because we've taken care in
                                * yy_get_next_buffer() to have set up
                                * yytext, we can now set up
                                * yy_c_buf_p so that if some total
                                * hoser (like flex itself) wants to
                                * call the scanner after we return the
                                * YY_NULL, it'll still work - another
                                * YY_NULL will get returned.
                                */
                               yy_c_buf_p = yytext_ptr + YY_MORE_ADJ;

                               yy_act = YY_STATE_EOF(YY_START);
                               goto do_action;
                               }

                       else
                               {
                               if ( ! yy_did_buffer_switch_on_eof )
                                       YY_NEW_FILE;
                               }
                       break;
                       }

               case EOB_ACT_CONTINUE_SCAN:
                       yy_c_buf_p =
                               yytext_ptr + yy_amount_of_matched_text;
```

```
                        yy_current_state = yy_get_previous_state();

                        yy_cp = yy_c_buf_p;
                        yy_bp = yytext_ptr + YY_MORE_ADJ;
                        goto yy_match;

                    case EOB_ACT_LAST_MATCH:
                        yy_c_buf_p =
                        &yy_current_buffer->yy_ch_buf[yy_n_chars];

                        yy_current_state = yy_get_previous_state();

                        yy_cp = yy_c_buf_p;
                        yy_bp = yytext_ptr + YY_MORE_ADJ;
                        goto yy_find_action;
                    }
                break;
                }

        default:
                YY_FATAL_ERROR(
                    "fatal flex scanner internal error--no action found" );
        } /* end of action switch */
                } /* end of scanning one token */
        } /* end of yylex */


/* yy_get_next_buffer - try to read in a new buffer
 *
 * Returns a code representing an action:
 *      EOB_ACT_LAST_MATCH -
 *      EOB_ACT_CONTINUE_SCAN - continue scanning from current position
 *      EOB_ACT_END_OF_FILE - end of file
 */

static int yy_get_next_buffer()
        {
        register char *dest = yy_current_buffer->yy_ch_buf;
        register char *source = yytext_ptr;
        register int number_to_move, i;
        int ret_val;

        if ( yy_c_buf_p > &yy_current_buffer->yy_ch_buf[yy_n_chars + 1] )
                YY_FATAL_ERROR(
                "fatal flex scanner internal error--end of buffer missed" );

        if ( yy_current_buffer->yy_fill_buffer == 0 )
                { /* Don't try to fill the buffer, so this is an EOF. */
                if ( yy_c_buf_p - yytext_ptr - YY_MORE_ADJ == 1 )
                        {
                        /* We matched a singled characater, the EOB, so
                         * treat this as a final EOF.
                         */
                        return EOB_ACT_END_OF_FILE;
                        }

                else
                        {
```

```
                        /* We matched some text prior to the EOB, first
                         * process it.
                         */
                        return EOB_ACT_LAST_MATCH;
                        }
                }

        /* Try to read more data. */

        /* First move last chars to start of buffer. */
        number_to_move = (int) (yy_c_buf_p - yytext_ptr) - 1;

        for ( i = 0; i < number_to_move; ++i )
                *(dest++) = *(source++);

        if ( yy_current_buffer->yy_buffer_status == YY_BUFFER_EOF_PENDING )
                /* don't do the read, it's not guaranteed to return an EOF,
                 * just force an EOF
                 */
                yy_n_chars = 0;

        else
                {
                int num_to_read =
                        yy_current_buffer->yy_buf_size - number_to_move - 1;

                while ( num_to_read <= 0 )
                        { /* Not enough room in the buffer - grow it. */
#ifdef YY_USES_REJECT
                        YY_FATAL_ERROR(
"input buffer overflow, can't enlarge buffer because scanner uses REJECT" );
#else

                        /* just a shorter name for the current buffer */
                        YY_BUFFER_STATE b = yy_current_buffer;

                        int yy_c_buf_p_offset =
                                (int) (yy_c_buf_p - b->yy_ch_buf);

                        if ( b->yy_is_our_buffer )
                                {
                                int new_size = b->yy_buf_size * 2;

                                if ( new_size <= 0 )
                                        b->yy_buf_size += b->yy_buf_size / 8;
                                else
                                        b->yy_buf_size *= 2;

                                b->yy_ch_buf = (char *)
                                        /* Include room in for 2 EOB chars. */
                                        yy_flex_realloc( (void *) b->yy_ch_buf,
                                                         b->yy_buf_size + 2 );
                                }
                        else
                                /* Can't grow it, we don't own it. */
                                b->yy_ch_buf = 0;

                        if ( ! b->yy_ch_buf )
```

```
                        YY_FATAL_ERROR(
                        "fatal error - scanner input buffer overflow" );

                    yy_c_buf_p = &b->yy_ch_buf[yy_c_buf_p_offset];

                    num_to_read = yy_current_buffer->yy_buf_size -
                                        number_to_move - 1;
#endif
                    }

                if ( num_to_read > YY_READ_BUF_SIZE )
                    num_to_read = YY_READ_BUF_SIZE;

                /* Read in more data. */
                YY_INPUT( (&yy_current_buffer->yy_ch_buf[number_to_move]),
                    yy_n_chars, num_to_read );
                }

        if ( yy_n_chars == 0 )
                {
                if ( number_to_move == YY_MORE_ADJ )
                    {
                    ret_val = EOB_ACT_END_OF_FILE;
                    yyrestart( yyin );
                    }

                else
                    {
                    ret_val = EOB_ACT_LAST_MATCH;
                    yy_current_buffer->yy_buffer_status =
                        YY_BUFFER_EOF_PENDING;
                    }
                }

        else
                ret_val = EOB_ACT_CONTINUE_SCAN;

        yy_n_chars += number_to_move;
        yy_current_buffer->yy_ch_buf[yy_n_chars] = YY_END_OF_BUFFER_CHAR;
        yy_current_buffer->yy_ch_buf[yy_n_chars + 1] = YY_END_OF_BUFFER_CHAR;

        yytext_ptr = &yy_current_buffer->yy_ch_buf[0];

        return ret_val;
        }


/* yy_get_previous_state - get the state just before the EOB char was reached */

static yy_state_type yy_get_previous_state()
        {
        register yy_state_type yy_current_state;
        register char *yy_cp;

        yy_current_state = yy_start;

        for ( yy_cp = yytext_ptr + YY_MORE_ADJ; yy_cp < yy_c_buf_p; ++yy_cp )
                {
```

```
		register YY_CHAR yy_c = (*yy_cp ? yy_ec[YY_SC_TO_UI(*yy_cp)] : 1);
		if ( yy_accept[yy_current_state] )
			{
			yy_last_accepting_state = yy_current_state;
			yy_last_accepting_cpos = yy_cp;
			}
		while ( yy_chk[yy_base[yy_current_state] + yy_c] != yy_current_state )
			{
			yy_current_state = (int) yy_def[yy_current_state];
			if ( yy_current_state >= 76 )
				yy_c = yy_meta[(unsigned int) yy_c];
			}
		yy_current_state = yy_nxt[yy_base[yy_current_state] + (unsigned int) yy_c];
		}

	return yy_current_state;
	}


/* yy_try_NUL_trans - try to make a transition on the NUL character
 *
 * synopsis
 *	next_state = yy_try_NUL_trans( current_state );
 */

#ifdef YY_USE_PROTOS
static yy_state_type yy_try_NUL_trans( yy_state_type yy_current_state )
#else
static yy_state_type yy_try_NUL_trans( yy_current_state )
yy_state_type yy_current_state;
#endif
	{
	register int yy_is_jam;
	register char *yy_cp = yy_c_buf_p;

	register YY_CHAR yy_c = 1;
	if ( yy_accept[yy_current_state] )
		{
		yy_last_accepting_state = yy_current_state;
		yy_last_accepting_cpos = yy_cp;
		}
	while ( yy_chk[yy_base[yy_current_state] + yy_c] != yy_current_state )
		{
		yy_current_state = (int) yy_def[yy_current_state];
		if ( yy_current_state >= 76 )
			yy_c = yy_meta[(unsigned int) yy_c];
		}
	yy_current_state = yy_nxt[yy_base[yy_current_state] + (unsigned int) yy_c];
	yy_is_jam = (yy_current_state == 75);

	return yy_is_jam ? 0 : yy_current_state;
	}


#ifndef YY_NO_UNPUT
#ifdef YY_USE_PROTOS
static void yyunput( int c, register char *yy_bp )
#else
```

```
static void yyunput( c, yy_bp )
int c;
register char *yy_bp;
#endif
        {
        register char *yy_cp = yy_c_buf_p;

        /* undo effects of setting up yytext */
        *yy_cp = yy_hold_char;

        if ( yy_cp < yy_current_buffer->yy_ch_buf + 2 )
                { /* need to shift things up to make room */
                /* +2 for EOB chars. */
                register int number_to_move = yy_n_chars + 2;
                register char *dest = &yy_current_buffer->yy_ch_buf[
                                        yy_current_buffer->yy_buf_size + 2];
                register char *source =
                                &yy_current_buffer->yy_ch_buf[number_to_move];

                while ( source > yy_current_buffer->yy_ch_buf )
                        *--dest = *--source;

                yy_cp += (int) (dest - source);
                yy_bp += (int) (dest - source);
                yy_n_chars = yy_current_buffer->yy_buf_size;

                if ( yy_cp < yy_current_buffer->yy_ch_buf + 2 )
                        YY_FATAL_ERROR( "flex scanner push-back overflow" );
                }

        *--yy_cp = (char) c;


        yytext_ptr = yy_bp;
        yy_hold_char = *yy_cp;
        yy_c_buf_p = yy_cp;
        }
#endif  /* ifndef YY_NO_UNPUT */


#ifdef __cplusplus
static int yyinput()
#else
static int input()
#endif
        {
        int c;

        *yy_c_buf_p = yy_hold_char;

        if ( *yy_c_buf_p == YY_END_OF_BUFFER_CHAR )
                {
                /* yy_c_buf_p now points to the character we want to return.
                 * If this occurs *before* the EOB characters, then it's a
                 * valid NUL; if not, then we've hit the end of the buffer.
                 */
                if ( yy_c_buf_p < &yy_current_buffer->yy_ch_buf[yy_n_chars] )
                        /* This was really a NUL. */
```

```
                        *yy_c_buf_p = '\0';

                else
                        { /* need more input */
                        yytext_ptr = yy_c_buf_p;
                        ++yy_c_buf_p;

                        switch ( yy_get_next_buffer() )
                                {
                                case EOB_ACT_END_OF_FILE:
                                        {
                                        if ( yywrap() )
                                                {
                                                yy_c_buf_p =
                                                yytext_ptr + YY_MORE_ADJ;
                                                return EOF;
                                                }

                                        if ( ! yy_did_buffer_switch_on_eof )
                                                YY_NEW_FILE;
#ifdef __cplusplus
                                        return yyinput();
#else
                                        return input();
#endif
                                        }

                                case EOB_ACT_CONTINUE_SCAN:
                                        yy_c_buf_p = yytext_ptr + YY_MORE_ADJ;
                                        break;

                                case EOB_ACT_LAST_MATCH:
#ifdef __cplusplus
                                        YY_FATAL_ERROR(
                                        "unexpected last match in yyinput()" );
#else
                                        YY_FATAL_ERROR(
                                        "unexpected last match in input()" );
#endif
                                }
                        }
                }

        c = *(unsigned char *) yy_c_buf_p;/* cast for 8-bit char's */
        *yy_c_buf_p = '\0';     /* preserve yytext */
        yy_hold_char = *++yy_c_buf_p;


        return c;
        }


#ifdef YY_USE_PROTOS
void yyrestart( FILE *input_file )
#else
void yyrestart( input_file )
FILE *input_file;
#endif
```

```
	{
	if ( ! yy_current_buffer )
		yy_current_buffer = yy_create_buffer( yyin, YY_BUF_SIZE );

	yy_init_buffer( yy_current_buffer, input_file );
	yy_load_buffer_state();
	}


#ifdef YY_USE_PROTOS
void yy_switch_to_buffer( YY_BUFFER_STATE new_buffer )
#else
void yy_switch_to_buffer( new_buffer )
YY_BUFFER_STATE new_buffer;
#endif
	{
	if ( yy_current_buffer == new_buffer )
		return;

	if ( yy_current_buffer )
		{
		/* Flush out information for old buffer. */
		*yy_c_buf_p = yy_hold_char;
		yy_current_buffer->yy_buf_pos = yy_c_buf_p;
		yy_current_buffer->yy_n_chars = yy_n_chars;
		}

	yy_current_buffer = new_buffer;
	yy_load_buffer_state();

	/* We don't actually know whether we did this switch during
	 * EOF (yywrap()) processing, but the only time this flag
	 * is looked at is after yywrap() is called, so it's safe
	 * to go ahead and always set it.
	 */
	yy_did_buffer_switch_on_eof = 1;
	}


#ifdef YY_USE_PROTOS
void yy_load_buffer_state( void )
#else
void yy_load_buffer_state()
#endif
	{
	yy_n_chars = yy_current_buffer->yy_n_chars;
	yytext_ptr = yy_c_buf_p = yy_current_buffer->yy_buf_pos;
	yyin = yy_current_buffer->yy_input_file;
	yy_hold_char = *yy_c_buf_p;
	}


#ifdef YY_USE_PROTOS
YY_BUFFER_STATE yy_create_buffer( FILE *file, int size )
#else
YY_BUFFER_STATE yy_create_buffer( file, size )
FILE *file;
int size;
```

```
#endif
	{
	YY_BUFFER_STATE b;

	b = (YY_BUFFER_STATE) yy_flex_alloc( sizeof( struct yy_buffer_state ) );
	if ( ! b )
		YY_FATAL_ERROR( "out of dynamic memory in yy_create_buffer()" );

	b->yy_buf_size = size;

	/* yy_ch_buf has to be 2 characters longer than the size given because
	 * we need to put in 2 end-of-buffer characters.
	 */
	b->yy_ch_buf = (char *) yy_flex_alloc( b->yy_buf_size + 2 );
	if ( ! b->yy_ch_buf )
		YY_FATAL_ERROR( "out of dynamic memory in yy_create_buffer()" );

	b->yy_is_our_buffer = 1;

	yy_init_buffer( b, file );

	return b;
	}


#ifdef YY_USE_PROTOS
void yy_delete_buffer( YY_BUFFER_STATE b )
#else
void yy_delete_buffer( b )
YY_BUFFER_STATE b;
#endif
	{

	if ( ! b )
		return;

	if ( b == yy_current_buffer )
		yy_current_buffer = (YY_BUFFER_STATE) 0;

	if ( b->yy_is_our_buffer )
		yy_flex_free( (void *) b->yy_ch_buf );

	yy_flex_free( (void *) b );
	}


#ifndef YY_ALWAYS_INTERACTIVE
#ifndef YY_NEVER_INTERACTIVE
extern int isatty YY_PROTO(( int ));
#endif
#endif

#ifdef YY_USE_PROTOS
void yy_init_buffer( YY_BUFFER_STATE b, FILE *file )
#else
void yy_init_buffer( b, file )
YY_BUFFER_STATE b;
FILE *file;
```

```
#endif


	{
	yy_flush_buffer( b );

	b->yy_input_file = file;
	b->yy_fill_buffer = 1;

#if YY_ALWAYS_INTERACTIVE
	b->yy_is_interactive = 1;
#else
#if YY_NEVER_INTERACTIVE
	b->yy_is_interactive = 0;
#else
	b->yy_is_interactive = file ? (isatty( fileno(file) ) > 0) : 0;
#endif
#endif
	}


#ifdef YY_USE_PROTOS
void yy_flush_buffer( YY_BUFFER_STATE b )
#else
void yy_flush_buffer( b )
YY_BUFFER_STATE b;
#endif

	{
	b->yy_n_chars = 0;

	/* We always need two end-of-buffer characters.  The first causes
	 * a transition to the end-of-buffer state.  The second causes
	 * a jam in that state.
	 */
	b->yy_ch_buf[0] = YY_END_OF_BUFFER_CHAR;
	b->yy_ch_buf[1] = YY_END_OF_BUFFER_CHAR;

	b->yy_buf_pos = &b->yy_ch_buf[0];

	b->yy_at_bol = 1;
	b->yy_buffer_status = YY_BUFFER_NEW;

	if ( b == yy_current_buffer )
		yy_load_buffer_state();
	}


#ifndef YY_NO_SCAN_BUFFER
#ifdef YY_USE_PROTOS
YY_BUFFER_STATE yy_scan_buffer( char *base, yy_size_t size )
#else
YY_BUFFER_STATE yy_scan_buffer( base, size )
char *base;
yy_size_t size;
#endif
	{
	YY_BUFFER_STATE b;
```

```
	if ( size < 2 ||
		base[size-2] != YY_END_OF_BUFFER_CHAR ||
		base[size-1] != YY_END_OF_BUFFER_CHAR )
		/* They forgot to leave room for the EOB's. */
		return 0;

	b = (YY_BUFFER_STATE) yy_flex_alloc( sizeof( struct yy_buffer_state ) );
	if ( ! b )
		YY_FATAL_ERROR( "out of dynamic memory in yy_scan_buffer()" );

	b->yy_buf_size = size - 2; /* "- 2" to take care of EOB's */
	b->yy_buf_pos = b->yy_ch_buf = base;
	b->yy_is_our_buffer = 0;
	b->yy_input_file = 0;
	b->yy_n_chars = b->yy_buf_size;
	b->yy_is_interactive = 0;
	b->yy_at_bol = 1;
	b->yy_fill_buffer = 0;
	b->yy_buffer_status = YY_BUFFER_NEW;

	yy_switch_to_buffer( b );

	return b;
	}
#endif


#ifndef YY_NO_SCAN_STRING
#ifdef YY_USE_PROTOS
YY_BUFFER_STATE yy_scan_string( yyconst char *str )
#else
YY_BUFFER_STATE yy_scan_string( str )
yyconst char *str;
#endif
	{
	int len;
	for ( len = 0; str[len]; ++len )
		;

	return yy_scan_bytes( str, len );
	}

#endif


#ifndef YY_NO_SCAN_BYTES
#ifdef YY_USE_PROTOS
YY_BUFFER_STATE yy_scan_bytes( yyconst char *bytes, int len )
#else
YY_BUFFER_STATE yy_scan_bytes( bytes, len )
yyconst char *bytes;
int len;
#endif
	{
	YY_BUFFER_STATE b;
	char *buf;
	yy_size_t n;
```

```
	int i;

	/* Get memory for full buffer, including space for trailing EOB's. */
	n = len + 2;
	buf = (char *) yy_flex_alloc( n );
	if ( ! buf )
		YY_FATAL_ERROR( "out of dynamic memory in yy_scan_bytes()" );

	for ( i = 0; i < len; ++i )
		buf[i] = bytes[i];

	buf[len] = buf[len+1] = YY_END_OF_BUFFER_CHAR;

	b = yy_scan_buffer( buf, n );
	if ( ! b )
		YY_FATAL_ERROR( "bad buffer in yy_scan_bytes()" );

	/* It's okay to grow etc. this buffer, and we should throw it
	 * away when we're done.
	 */
	b->yy_is_our_buffer = 1;

	return b;
	}
#endif


#ifndef YY_NO_PUSH_STATE
#ifdef YY_USE_PROTOS
static void yy_push_state( int new_state )
#else
static void yy_push_state( new_state )
int new_state;
#endif
	{
	if ( yy_start_stack_ptr >= yy_start_stack_depth )
		{
		yy_size_t new_size;

		yy_start_stack_depth += YY_START_STACK_INCR;
		new_size = yy_start_stack_depth * sizeof( int );

		if ( ! yy_start_stack )
			yy_start_stack = (int *) yy_flex_alloc( new_size );

		else
			yy_start_stack = (int *) yy_flex_realloc(
					(void *) yy_start_stack, new_size );

		if ( ! yy_start_stack )
			YY_FATAL_ERROR(
			"out of memory expanding start-condition stack" );
		}

	yy_start_stack[yy_start_stack_ptr++] = YY_START;

	BEGIN(new_state);
	}
```

```
#endif


#ifndef YY_NO_POP_STATE
static void yy_pop_state()
	{
	if ( --yy_start_stack_ptr < 0 )
		YY_FATAL_ERROR( "start-condition stack underflow" );

	BEGIN(yy_start_stack[yy_start_stack_ptr]);
	}
#endif


#ifndef YY_NO_TOP_STATE
static int yy_top_state()
	{
	return yy_start_stack[yy_start_stack_ptr - 1];
	}
#endif

#ifndef YY_EXIT_FAILURE
#define YY_EXIT_FAILURE 2
#endif

#ifdef YY_USE_PROTOS
static void yy_fatal_error( yyconst char msg[] )
#else

static void yy_fatal_error( msg )
char msg[];
#endif
	{
	(void) fprintf( stderr, "%s\n", msg );
	exit( YY_EXIT_FAILURE );
	}



/* Redefine yyless() so it works in section 3 code. */

#undef yyless
#define yyless(n) \
	do \
		{ \
		/* Undo effects of setting up yytext. */ \
		yytext[yyleng] = yy_hold_char; \
		yy_c_buf_p = yytext + n - YY_MORE_ADJ; \
		yy_hold_char = *yy_c_buf_p; \
		*yy_c_buf_p = '\0'; \
		yyleng = n; \
		} \
	while ( 0 )


/* Internal utility routines. */

#ifndef yytext_ptr
```

```
#ifdef YY_USE_PROTOS
static void yy_flex_strncpy( char *s1, yyconst char *s2, int n )
#else
static void yy_flex_strncpy( s1, s2, n )
char *s1;
yyconst char *s2;
int n;
#endif
        {
        register int i;
        for ( i = 0; i < n; ++i )
                s1[i] = s2[i];
        }
#endif


#ifdef YY_USE_PROTOS
static void *yy_flex_alloc( yy_size_t size )
#else
static void *yy_flex_alloc( size )
yy_size_t size;
#endif
        {
        return (void *) malloc( size );
        }

#ifdef YY_USE_PROTOS
static void *yy_flex_realloc( void *ptr, yy_size_t size )
#else
static void *yy_flex_realloc( ptr, size )
void *ptr;
yy_size_t size;
#endif
        {
        /* The cast to (char *) in the following accommodates both
         * implementations that use char* generic pointers, and those
         * that use void* generic pointers.  It works with the latter
         * because both ANSI C and C++ allow castless assignment from
         * any pointer type to void*, and deal with argument conversions
         * as though doing an assignment.
         */
        return (void *) realloc( (char *) ptr, size );
        }

#ifdef YY_USE_PROTOS
static void yy_flex_free( void *ptr )
#else
static void yy_flex_free( ptr )
void *ptr;
#endif
        {
        free( ptr );
        }

#if YY_MAIN
int main()
        {
        yylex();
```

```
        return 0;
        }
#endif
#line 97 "paco.l"


yywrap(){return(1);}


int yyerror(char *s) {
    int i;
    unsigned int k;
    char sz[2];sz[1]='\0';

    free(outsrc);

    outsrc = strdup("");
    AddStr("\nputs(\"!!!Error in source part:\\n\");");

        /*
    AddStr("\nputs(\">>>***** ");
    for(k=0;k<strlen(s);++k){
        sz[0] = s[k];
        if(s[k]=='\"')
            AddStr("\\\"");
        else
            AddStr(sz);
    }
    AddStr(" *****:\");\n");
    */

    i = strlen(yy_current_buffer->yy_ch_buf)-20;
    i = ((i<0)?0:i);
    s = yy_current_buffer->yy_ch_buf+i;

    AddStr("\nputs(\"");
    for(k=0;k<strlen(s);++k){
        sz[0] = s[k];
        if(s[k]=='\"')
            AddStr("\\\"");
        else if(s[k]=='\n' || s[k]=='\r')
            AddStr("\\n");
        else
            AddStr(sz);
    }
    AddStr(" \\n\");\n");

    yyterminate();

    return 0;
}


int __PACOCompile(const char* insrcx, char** outsrcx) {
        char* bodystr;

    YY_BUFFER_STATE buf_state;
```

```
        num = 0;
        namestr=NULL;
    include_stack_ptr = 0;

    buf_state = yy_scan_string(insrcx);
    yy_switch_to_buffer(buf_state);

    outsrc = strdup("");
    argstr = strdup("");

    yyparse();

        bodystr = strdup(outsrc);
        free(outsrc);
        outsrc=strdup("");


    AddStr("#include <stdlib.h>\n");
    AddStr("#include <stdio.h>\n");
    AddStr("#include <string.h>\n");
    AddStr("#include <math.h>\n");
    AddStr("#include \"utl.h\"\n");

    AddStr("\nvoid main(void)\n");
    AddStr("{\n");
    AddStr("\n");

        AddVarDecl();
        AddStr("\n");

        AddStr(bodystr);

        AddStr("\n");
        AddStr("\n");
        ReleaseVar();
    AddStr("\n\n}\n");


    *outsrcx = (char*)malloc((strlen(outsrc) + 1)*sizeof(char));
    strcpy(*outsrcx,outsrc);


    free(argstr);
    free(outsrc);

        yy_delete_buffer(buf_state);
        removeall(&namestr);

        free(bodystr);

    return 0;
}

int __PACOCompileRelease(char** rtntxt) {
    if(*rtntxt!=NULL){
         free(*rtntxt);
         *rtntxt=NULL;
    }
```

```
    return 0;
}

void inputstr()
{
        char str[800];

        printf("\nStart to input program from here: \n[type \"end\" in a new line to end]\n\n");
        gets(str);

        while(strcmp(str,"end")!=0){
                in = (char*)realloc(in, (strlen(in) + strlen(str) + 2)*sizeof(char));
                strcat(in, str);

                strcat(in,"\n");
                gets(str);
        }

        //free(str);
//printf(in);

}

void main( void )
{
    /* yyin = fopen( argv[0], "r" ); */
        //int i;
        //for(i=0;i<10000;++i){
        char *out;
        FILE *fout;


        //in=strdup("y=x=2+8;while(i<10){x=x+1;} x=(s<9);");
        //in=strdup("for(i=0;i<10;i=i+1)for(j=0;j<10;j=j+1){x=x+1;r=0;}");
        //in=strdup("y=x=2+(8+9);x=f(1,3);");
        //in=strdup("x=[1];");
        //in=strdup("x=[1,2,3];");
        //in=strdup("x=[1,2;3,4];d=6;print(d);");
        //in=strdup("x=12;d=6+4-9*2;print(d);");
        //in=strdup("x=[1,2,3;4,5,6];print(x);");
        //in=strdup("                                                                    ");

        //in = (char*)strdup(" x=123;");
        //in=malloc(8*sizeof(char));   ////////////////////////////////
        //in[0]=0;

        in=strdup("pi=3.14159;\n                                \n");
        inputstr();

        //printf(in);

if(strlen(in)>0){
        __PACOCompile(in, &out);

        //printf("%s\n",out);

        fout=fopen("pf.c","w");
```

```
        fprintf(fout,out);

        fclose(fout);

        __PACOCompileRelease(&out);
}

        //}
        free(in);
        //getchar();

}
```