1991

Apt Compiler Toolkit User Manual

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Ufuk Verun

Apt Compiler Toolkit

George K. Thiruvathukal
Tellabs Incorporated
Data Communications Division

Illinois Institute of Technology
Department of Computer Science
Internet: gkt@iitmax.iit.edu

Ufuk Verun
Illinois Institute of Technology
Department of Computer Science
Internet: thssuxv@iitmax.iit.edu

1.0 Overview of the Apt Compiler Toolkit

1.1 Introduction

The Apt Compiler Toolkit was designed to address the need for structured, efficient, portable, and capable tools to prototype language translators and compilers. In the current release of the toolkit tools are available for the generation of scanners, parsers, and data structures. A robust library of functions is supplied with the toolkit which includes support for the scanner, the parser, abstract data types (which are commonly used in language translators/compilers), and string functions.

1.2 The Tools

The following subsections provide a synopsis of the supported tools in the Apt Compiler Toolkit. As we have dedicated a chapter for the discussion of each tool, we will defer discussion of details to the respective chapters.

1.2.1 The Apt Parsing Tool (APT)

The Apt Parsing Tool (APT) is a tool for the generation of LL(1) Parsers. The tool and the code it produces has been designed to be efficient, portable, structured, language-independent, and human-readable. We urge the reader to examine the output of APT (and the other tools) and compare its readability to other popular tools (like yacc). The code produced has the appearance of being written by hand.

1.2.2 The Apt Scanning Tool (AST)

The Apt Scanning Tool is a tool for the generation of scanners based on deterministic finite state automata (DFA). While we appreciate the elegance of regular expressions, many experience compiler writers and theoreticians are in agreement that regular expressions, when translated to optimized DFA, generally incur a larger space overhead than hand-coded DFA. With the Apt Scanning Tool one can specify the lexical elements as an FSA in symbolic fashion. Extensive library support is provided to support a variety of common transition character classes and actions which must be performed during scanning. As is the case with APT, AST has been designed to accommodate the same features as APT. AST and APT can be integrated to build a recognizer for virtually any programming language of practical interest.

1.2.3 The Apt Node Tool (ANT)

The Apt Node Tool is a tool which generates code for many typically required manipulations of data structures. Given a set of nodes (for instance, an Expression), ANT can be used to generate code for the allocation, disposal, modification, shallow replication, deep replication, and (a skeleton of) general traversal of the type. The tool was developed initially as an ACT demonstration program. After the first release it was considered so useful to a number of projects that it was refined into a tool to support all of the common operations needed for the management of data structure classes in a compiler. The ANT can work with AST and APT to build a recognizer and all data structures (effectively the compiler front-end) so the compiler writer can work on code generation (of which we hope to assuage by supporting with tools someday).

1.2.4 The Apt Data Type (ADT) Library

Included with the Apt Compiler Toolkit is a robust collection of generic abstract data types. The types have been carefully chosen to account for all of the possible data structures a compiler writer might need to construct in a compiler. Included in the ADT Library are static stacks, dynamic stacks, queues, deques, AVL trees, tables, hash tables, buffers, and buffered input streams. While we plan to augment the library with more types (to accommodate other users who might find the libraries useful, like database designers), we believe our library adequately addresses the needs of the compiler writing community. A substantial section of the manual is dedicated to the ADT Library.

1.2.5 How to Read and Use this Document

We have designed the document to “jump start” the compiler writer in you. The discussions of the tools and libraries are deliberately succinct. We will not present any examples in the chapters on tools and libraries and will allocate an entire chapter to present a non-trivial example of a translator which uses APT, AST, ANT, and the ADT Library.

While we hope the document is sufficiently interesting and concise to maintain your interest, we understand that the nineties are a time when people hardly have time for reading literature (especially of the technical variety). If you just do not have the time to

read the sections on the tools, you can skip the sections on the tools and libraries and proceed to the detailed example (which is self-contained).

2.0 The Apt Parsing Tool

2.1 Introduction

The Apt Parsing Tool is a tool for the generation of LL(1) parsers. As input, it accepts a specification file which contains the language grammar, the prototypes for the semantic actions, error recovery information, and alias information. We will discuss each of these aspects in detail in the section pertaining to the specification file.

The language grammar which is acceptable to APT must be in LL(1) form. If the grammar is not in LL(1) form, APT will make the determination and inform the user of how the grammar violates the LL(1) property. Grammars in which left recursion, ambiguity, or common prefixes are present typically violate the LL(1) property.

Semantic analysis is specified at a high-level in the specification file. The decision to do this was motivated by our desire to achieve language-independence in our tools. A semantic action is specified in a syntax which permits a complete function prototype to be defined and what semantic stack elements are passed to the semantic action when it is invoked.

Error recovery information merely involves the specification of a set of fiducial symbols which can be used to attempt recovery from syntax errors. We are presently exploring more advanced techniques for the management of errors and intend to expand APT once we have determined which technique is well-suited to LL(1) parsing. From our own experience syntactic error recovery is really not so important and often leads to poor diagnosis and recovery. A fast parser and intelligent first error message tend to be most useful for one to get a program compiled in an expedient manner.

Alias information enables the user of APT to specify the grammar exactly as he or she has written it and then associate symbols which are not identifiers with identifiers. This feature must be used with grammars containing non-identifier symbols to ensure code generation of symbols which are identifiers.

The abovementioned features will all be discussed in detail in the section pertaining to the specification file syntax. In the next section we will discuss some of the features of APT and (hopefully) convince you that there are compelling reasons to use it and avoid many of the other available parsing tools.

2.2 What in the World is so Good about APT?

With the myriad of software tools and technology out there one has to at least ponder the issue of why yet-another-parsing-tool ought to be used. As one of the authors once had to make a case for using it to develop a compiler at a former place of employment, the

following paragraphs are somewhat of a regurgitation and an expansion of reasons for using the Apt Parsing Tool (as well as the Apt Compiler Toolkit)

2.2.1 Efficienct

A claim often made by people and companies who manufacture tools, we have designed APT with efficiency in mind. The tool generates efficient table strcutures for efficient execution of the parser and the semantics. The parse tables generated are compressed for efficient representation and access. Semantic actions are represented by a jump table.

2.2.2 Structured

The techniques used to develop the tool, its libraries, and the companion tools are structured and object-oriented. In most of the parsing tools we have studied and used the lexical analysis, parsing, and semantics are combined. There is certainly nothing structured about such a design. The APT philosophy is for these phases to be logically divided among a number of modules. We say “a number of modules,” as the user can define his or her semantics as a number of modules. The APT specification files does not permit any coupling of parsing with lexical analysis or semantics. These activities must be defined in separate compilation units.

2.2.3 Type-Safe Semantics

Semantic analysis is conducted in a type-safe manner. The term type-safe was emanated from the C/C++ world when mechanisms were added to the C language to ensure that the linkage of object modules resulted in a warning or an error, if a mismatch occurred between a function declaration and its usage. In APT a similar mechanism exists for semantics. Once the interface to a semantic routine is defined in the specification file, the semantic routine must be defined exactly as specified. When the semantics are being executed, APT checks whether the attributes from the stack are of the correct types before passing them to a semantic routine. While type-safety does not preclude errors from occurring, it does assuage the process of debugging. If one knows that a routine was called correctly (with incorrect data), the incorrect data can minimally be browsed. As all C programmers know, it is not always possible to determine what is being referenced in a pointer context.

2.2.4 Debugging Support

Let us face the music. Even the best programmers do not write perfect code all the time. Because we realize the users of our tools to be confined to the human race, APT has been designed to include extensive debugging support. The scanner, parser, and semantics can all be tested without using a debugger, as APT has several trace options available. These trace options can be imported from the APT library and conditionally invoked to test various aspects of the compiler or translator being developed.

2.2.5 Language Independence

All of the tools in the Apt Compiler Toolkit are designed to be as language independent as possible. What this means is that the specification file syntax is independent of a particular language. This does not mean that we have not used a particular language for

the implementation of ACT and its libraries. We firmly believe in our decision to maintain language independence and believe the decision enables the language designer and implementor to avoid the details of coding in a specific language until the design of the language and semantics is completed.

2.2.6 Integration

APT integrates well with other tools, like lex, which are designed for lexical analysis. It also integrates with a number of tools found in the Apt Compiler Toolkit. Coupled with ACT, syntax analysis and construction-oriented semantics can be achieved without the need to write a single line of code.

2.2.7 Multiple Semantic Phases

APT is perhaps the only documented tool to provide support for multiple semantic phases. While we are still experimenting with the best syntax for defining semantics over multiple phases, a useful (but tentative) form of it exists in the current version of APT. This feature is vital to the correct implementation of a programming language which contains forward references, such as Modula-2. To implement such a feature the semantics must be divided into two phases: declaration analysis and expression analysis. The idea is to spend effort in one phase to collect declarations and install them in the symbol table and then spend effort in the next phase to construct data structures for expressions and statements. We have used the feature of multiple semantic phases for the development of an Icon compiler (which is still under development).

2.2.8 Portable

Many of the compiler tools available on the market and in the public domain are severely weak with respect to the feature of portability. Our tools were designed to be portable from the start. The APT tools can be built on any platform which supports Portable or ANSI C language compilers. The code generated by the tools and the ACT libraries can be compiled with any C compiler (including the ones which accept ancient dialects of C). As we have a need for the translators we develop to work on a variety of platforms, APT has literally paved the way for it to be possible. We gave up on yacc, because it does not generate acceptable code for ANSI C compilers. The Apt Compiler Tools have been compiled on many UNIX systems, DOS, and AmigaDOS. We have not had the software to test other platforms, but we have good reasons to believe it will work when we do!

2.2.9 Intelligible

A word which means “able to be understood or comprehended,” the ACT tools have been designed to contain and to produce readable source code. Such code can be adapted by users, if necessary. Other tools produce code which appears to be some random permutation of the digits of pi interspersed with random alphanumeric characters. The code files produced by the ACT tools produce code which looks better than code which is written by hand. Comments and annotations are inserted in the code, when necessary, to enhance readability and comprehension.

2.3 The Specification File

The specification file is used to define the syntax and semantics of a programming language save the implementation details of the semantics. A specification file is divided into one or more sections. A section is opened via a section keyword, which begins with a percent sign and is followed by PRODUCTIONS, ACTIONS, ALIASES, or FIDUCIALS. If one of these names does not appear after the percent sign, a section is not opened and the symbol is not considered a special keyword. A section may appear more than one time. When more than one occurrence of a section is detected, the items which appear in it are appended to lists of items detected in previous occurrences of the same type of section. This feature allows the language designer the flexibility to specify related grammatical rules and semantics alternatively, which is useful for large and complicated programming languages.

2.3.1 %PRODUCTIONS Section

The %PRODUCTIONS section contains is list of lines, each of which is a production with the following syntax:

<production> ::= <lhs> -> <rhs>
<lhs> ::= <symbol>
<rhs> ::= { <symbol> }*
<symbol> ::= { <printable-character> }+

While we are assuming the reader is familiar with BNF grammars, we wish to comfort the reader with some explanation in his or her most familiar language. A production is defined above to be a left-hand-side followed by a right-hand-side. The left-hand-side must be a single symbol (called a nonterminal symbol in compiler literature); the right-hand-side is zero or more symbols in length. One must separate the symbols (as well as the arrow which delineates the left and right-hand-sides by white space. A symbol is defined to be a sequence of one or more printable characters. The definition of a printable character is somewhat standardized; however, check the C macro (shipped with your C compiler) for the macro “isprint,” as this is the macro we use in APT to determine whether a candidate character for a symbol is printable.

The symbols which appear on the right-hand-side are either terminal or nonterminal in nature. Any symbol which appears on the left-hand-side is deemed nonterminal by APT; any other symbol is terminal. A terminal symbol can either correspond to a token (which is obtained from the lexical analyzer) or a phrase marker. To define a terminal symbol as a token does not require any effort at all; however, to define a terminal symbol as a phrase marker requires one to define it as an action in the %ACTIONS section.

2.3.2 %ACTIONS Section

The %ACTIONS section consists of the section header, followed by a list of phase names (see %PHASES) on the same line, followed by a list of action definitions.

At least one phase name must be listed per %ACTIONS section. Every translator has at least one semantic phase. If the language designer does not perceive a need for more than one semantic phase, it is suggested that he name the phase after the language.

An action definiton is an association between a terminal symbol (called a phrase marker) and a function. The function is declared in a special manner so the language designer can tell APT what semantic stack elements are passed to the semantic routine when invoked. The following syntax is used to define a phrase marker:

<semantic-action> ::= <phrase-marker> <function>
<phrase-marker> ::= <symbol>
<function> ::= <function-name> (<arguments>) <return>
<arguments> ::= { <argument> }*
<argument> ::= <disp> : <type-name>
<disp> ::= <non-negative-integer>
<non-negative-integer> ::= { <digit> }+
<return> ::= : (<type-name>, <pop-count>)
<pop-count> ::= <non-negative-integer>
<type-name> ::= <identifier>
<function-name> ::= <identifier>
<symbol> ::= { <printable-character> }+

The above is looks intimidating, but it is really simple. A semantic action is defined as an association between a phrase marker (a symbol) and a function. The function is really a function prototype, which is specified as a function name followed by a list of arguments (possibly empty), followed by return information. Each argument is a colon separated pair of a non-negative integral displacement and a type (which must be an identifier). Here the displacement refers to the position at which the attribute to be passed to the semantic routine is located along the semantic stack (with the top of stack growing downward) and the type-name is type of the attribute. The return information is a comma-separated pair of a type-name and a pop-count. The type name is the return type of the semantic function. The pop-count specifies the number of symbols to be popped from the semantic stack before the return result is pushed.

The reader might be confused at this point, so we will provide an example of concrete nature. Suppose we defined three productions as follows:

%PRODUCTIONS

Expression -> Term Expression’
Expression’ -> op Term _BinOp Expression’
Expression’ ->

%ACTIONS Equation

_BinOp ExpressionBinaryNew(3:ExpRec,2:Token,1:ExpRec):(ExpRec,3)

We first note that the above is not a complete specification for the familiar expression language. It merely specifies enough for us to illustrate how a terminal symbol is mapped to a phrase marker and a phrase marker is associated with a function (which must either be generated or written by the language implementor).

In the above grammar the nonterminal symbols are Expression and Expression’; the terminals are op and _BinOp. The %ACTIONS section is where an association is made between the terminal symbol _BinOp and a function ExpressionBinaryNew. The function ExpressionBinaryNew is a function (user-defined) which takes a total of three

arguments: the left operand (an ExpRec), the operator (a Token, which is a type built-in to the Apt Compiler Toolkit), and the right operand (an ExpRec). It returns an ExpRec (which is a synthesis of the three values passed as parameters) which is pushed onto the semantic stack after three symbols are popped.

We apologize if this example has the appearance of being deliberately terse, but we believe the final section pertaining to the design and implementation of a complete translator will elucidate the points above and provide a concrete example of how a translator for a language of simple to intermediate difficulty is developed.

2.3.3 %ALIASES

Aliases are really not essential to language design, but they are essential to APT itself. One of the features of the APT specification file is that one can use non-alphanumeric (but printable) character strings to represent grammar symbols. Because APT produces readable code, however, it needs to have a valid identifier associated with the symbol, if the symbol is not a valid identifier for the target programming language (which is C or Modula-2 at the moment). The syntax for an alias is stated simply:

<alias> ::= <symbol> <identifier>

The above rule specifies that a symbol is associated with an identifier. Aliases are not transitive, as we cannot see much point in supporting the notion.

2.3.4 %FIDUCIALS

Fiducial symbols can be specified for panic mode error recovery. A fiducial symbol is a symbol which can be trusted to appear in the input and lead to proper resynchronization of the parser, if an error occurs. The syntax for the specification of fiducials is simpler than that for any other feature:

<fiducial> ::= <symbol>

We are planning on adding advanced error recovery capabilities to APT. As an exercise, the reader is invited to implement an error recovery feature in APT. Since he or she has the source code at his or her disposal, the opportunity to do so is ripe.

2.3.5 %PHASES

Phases may be specified when it is desired for semantics to be executed on the stream of syntactically correct input several times. All phrase markers must be defined for each phase listed. The syntax for the specification of phases is trivial:

<phase> ::= <identifer>

It is important to include one or more phase identifiers after the %ACTIONS section header to inform APT that the actions are defined for the listed phase names. As mentioned previously, multiple %ACTIONS sections may be specified, so each semantic phase can be defined in terms of one (or more) actions section(s).

In a future release of the documentation we will provide a detailed example of how multiple semantic phases are employed. For now, however, we leave the reader with the definition of the feature and his or her imagination for how it might be used.

2.4 Command Line Interface

2.4.1 Command Line Interface

The interface to the Apt Parsing Tool is command line oriented. The figure below illustrates the screen of text which appears when you invoke the Apt Parsing Tool with no command line arguments. As is the protocol for most Unix tools, a tool invoked with no command line arguments or with incorrect arguments greets you with a usage screen from which you will (hopefully) be able to figure out the proper syntax of the command line required to invoke the tool. We have not adhered to tradition exactly, however, as the Apt Parsing Tool usage screen is a bit verbose (compared to Unix tools) and much more helpful. We do not feel this departure from tradition is disadvantageous to the reader.

Apt Parsing Tool 3.0
Copyright (c) 1991 -- Apt Technologies

Apt Parsing Tool -- ANSI-C Language Edition

An APT command line uses the following syntax:

% apt <file-name> <options>

where:

<file-name> ::= <file-prefix>.grm
<options> ::= { <option> }
<option> ::= -a<switch> (Analyze grammar only (-c- and -l- are implicit))
<option> ::= -c<switch> (Code generation (default enabled))
<option> ::= -i <header-prefix-list> (Include type files)
<option> ::= -l<switch> (List file (default enabled))
<option> ::= -o <code-prefix> (Generate stub actions)
<option> ::= -s<switch> (Semantic stack checks (default enabled))
<option> ::= -t<switch> (Terminal symbols (default enabled))
<option> ::= -w<switch> (Generate code on warnings (default disabled))
<switch> ::= + | - | null

The Apt Parsing Tool accepts as input a grammar file followed by a number of options. These options are each explained below:

- -a : analyze grammar only

Option a tells the Apt Parsing Tool to perform grammatical analysis only. This implies that no code generation or list file generation takes place. The grammatical analysis is

useful to determine whether there are any problems with the input grammar (such as left recursion, infinite-deriving nonterminals, unreachable symbols, and et cetera) before any code generation occurs. By default this option is disabled.

- -c : code generation

Option c tells the Apt Parsing Tool to disable code generation. It might be useful to do this when there is a need to observe the results of the LL(1) table construction phase. Sometimes an LL(1) problem requires the user to explore the FIRST and FOLLOW sets to understand the nature of it. Since option c does not disable option l, the user can obtain a list file without a code file being generated.

- -i : list of files to be included

Option i permits the user to have a list of files be included by the code and header files of APT. This option is import so the user can convey type information to APT. As the specification file constrains the user to identifier type names, the type names must be defined somewhere (like a C header file). It is not mandatory that this be done; however, to achieve the maximum level of type checking, one ought to define every type name listed in the specification file in a header file.

- -l : list file generation

Option l tells the Apt Parsing Tool to disable list file generation. Once the language definition is believed to be correct, there is no reason to generate a list file, as its primary utility is to determine whether there are problems with the language grammar.

- -o : generate stub action functions

Option o is very useful. Never forget it. Once the language description and phrase markers are all defined, the user can use this option to have Apt generate a file of stub action functions. By generating a file of stub actions, the lexical analyzer, the parser, and the stub action functions can be linked together to produce a syntax analyzer for the language being implemented. The user can test whether the semantics will execute correctly (given a suite of programs written in the language) without actually executing any semantics code. When he or she is confident that the semantics are being called correctly and the semantic stack is being left in a healthy state, the actual semantic functions can be implemented and tested. This approach to development is much simpler than the one which is required for other parsing tools (such as yacc), where the correctness of semantics cannot be tested at all. Also, this approach is consistent with the object (or modular) paradigm, where objects (or modules) are tested for their workability and then incorporated into the big picture.

- -s : disable semantic stack checks

Option s disables the generation of run-time code to ensure that attributes pulled from the semantic stack each are of the type required by the semantic action function (declared in the %ACTIONS section). Generally, one need not use this option, as the run-time code can be suppressed when the code file is compiled. The Makefile (distributed with the Apt system) explains how one can disable the semantic error checking code. The bottom line: use this option only if the code size is exorbitant.

- -t : disable generation of terminal symbol enumeration type

Option t disables the generation of an enumeration type for the terminal symbols of the language. It is necessary for the user to do this when he or she supplies his or her own definitions to APT in a header file. If the user opts to use the Apt Scanning Tool, he or she need never be concerned with this option. If this option is used, it is the responsibility of the user to ensure the file containing terminal symbol #defines or enumeration type(s) is supplied to APT (via option i). Further, the values of these symbols must not conflict with the values of other symbols defined by APT. The bottom line: use the Apt Scanning Tool to generate a scanner for use by the APT parser.

- -w : generate code on warnings

Option w enables code to be generated when there are warnings which result from grammatical analysis or LL(1) table construction. Use this option when the LL(1) table conflicts are well-understood (for instance, the if-then-else problem is a well-understood parsing table conflict which is resolved in favor of a shift).

2.4.2 Output Files

A number of output files are potentially produced as a result of an invocation of APT with a grammar file and appropriate options. We present each one below with a high-level description of its purpose. The details of their contents are not relevant to this document and will be the subject of a detailed technical report. We believe the user can browse these files and understand their contents, as the code produced is very readable and almost self-documenting. Assume the grammar file is named language.grm.

- language.c

This is the code file. It contains the parse table (prediction table), the semantic actions tables (which consist of a phase table and an action table per phase), phrase functions, and index tables (to expedite lookups on the parse table. The parse table is compressed to minimize the amount of space required for its representation. The phrase functions contain inline code to invoke the associated semantic action function with the appropriate stack elements, to consume attributes from the semantic stack, and to push a synthesized attribute onto the semantic stack.

- language.h

This is the header file. It contains export information for the code file. It also contains enumeration types for the terminals, the nonterminals, and the actions.

- language.td

This is the type definition file. It contains default type definitions for the type names which appear in the %ACTIONS section. Each of these type names is defined to be a generic pointer (which, in C, is a pointer to void - void *). When the user specifies his or her own type definitions (via option i), the default type definitions are suppressed and replaced with the user type definitions. This is reflected in the file via a replacement of the type definitions with a #include (or import directive/syntax appropriate to the target language).

- language.pt

This is the prototypes file. This file is important in the context of the C programming language, as there are more than subtle differences in how the interface to a function is defined from old standards to new standards.

- language.lst

This is the listing file. It contains all information pertinent to grammatical analysis (in the context of LL(1) grammars): FIRST, LAST, NEXTTO, and FOLLOW sets. Other information includes the minimum sentence derivable from a nonterminal (including infinity), reachability information, the symbol table, and the production table. It is very useful for determining whether a problem exists with the language from an LL(1) perspective.

- language.err

This is the error file. Included here is information about errors which arise from the generation of the LL(1) parse table. If a conflict occurs during the process of generating the parse table, an informative message will appear here, along with an explanation of what type of conflict it is (whether it arose from a FIRST set or a FOLLOW set) and how it was resolved. A resolution is always made in favor of the production which appeared first. In a later version we might add support for syntax which permits the specification of how resolution is done for a given set of nonterminals, but the current mechanism adequately solves the problem with minimal burden for the user.

3.0 The Apt Scanning Tool

3.1 Introduction and Overview of Features

The Apt Scanning Tool is a tool for the generation of a scanner (a function) to achieve lexical analysis. As input, it accepts a specification file which contains a description of a deterministic finite state machine (DFSM). The format of the specification allows the DFSM to be specified as symbolically as possible.

Also provided by the tool is extensive support for character classes and intrinsic lexical actions, both of which enable the language implementor to specify a DFSM in a terse manner. Character classes can be used to label a transition from one state to another state, which eliminates the need to replicate common transitions from one state to another state. Intrinsic lexical actions are present to perform various lexical actions. Some of the traditional lexical actions include the following:

- match the current character and put it into the buffer which holds the token.
- same as 1, but advance the input pointer once done with 1.
- ignore the current character and advance the input pointer, as in 2.
- push the current character back into the input stream and do not add it to buffer.
- like 4, but push back the last character which entered the token buffer.
- like 5, but push back the second to last character which entered the token buffer.

We believe these cases encompass all lexical actions appropriate to modern programming languages. Additional lexical actions can be written by the user, and daisychaining can occur with the existing lexical actions.

The fundamental design principle of the Apt Scanning Tool was simplicity, efficiency, and elegance. By elegance we do not insinuate the notion of theoretical elegance; we were more concerned about structural (and even an object-oriented) elegance. A scanner developed with AST has components which can directly be reused. We even provide a library of rules which manifest the collection of different lexical elements (identifiers, floating point numbers, the Pascal subrange operator [which is ..], various comment styles, operators, and et cetera). The user of the Apt Scanning Tool can design a scanner within minutes for most practical (and well-designed from the lexical perspective) programming languages.

3.2 Specification File Syntax

As alluded to in the overview, the specification file for AST permits the simple and concise specification of the deterministic finite state machine in a symbolic and intuitive manner. The line-oriented syntax in EBNF follows:

<line> ::= <rule>
<rule> ::= <source-state> <transition-label> <dest-state> <action> [<attribute>]
<source-state> ::= <state>
<dest-state> ::= <state>
<transition-label> ::= <c-identifier> | <c-character-string>
<action> ::= <identifier>
<attribute> ::= <identifier>

A brief discussion of the above grammar is warranted. Each line consists of a rule. A rule is defined to be a transition from a source state to a destination state. The transition is either an identifier or a character string (both as defined in the C programming language). If a transition is encoded as an identifier, the identifier corresponds to an intrinsic character class (described below); otherwise, the transition applies to each character in the character string. The action, an identifier, is either an intrinsic or user-defined lexical action. As we will discuss later, the user-specified lexical action must be either added to the Apt Scanning Tool library or placed in a separate file which is linked with the scanner (or translator) being built. An attribute may optionally be specified. This attribute implies that a token is returned by the lexical analyzer in the destination state. The destination state is determined to be an accepting state. Transitions from a state which returns an attribute are undefined, and will be ignored by the Apt Scanning Tool.

The start state is determined to be the first source state (i.e. the source state listed on the first line). Accepting states are determined to be the source states which happen to list an attribute to be returned. It is possible for a state to be an accepting one and yet have transitions to other states.

3.2.1 Intrinsic Character Classes

LowerAlpha - a lower case character in the range ‘a’ through ‘z’

UpperAlpha - an upper case character in the range ‘A’ through ‘Z’
Alpha - either LowerAlpha or UpperAlpha
AlphaNumeric - either Alpha or Numeric
Numeric - a character in the range ‘0’ through ‘9’
Default - any character
Lambda - any character
EndOfFile - the end-of-file character
HexDigit - any Numeric or character in the range ‘A’ through ‘F’
OctDigit - any Numeric except ‘8’ and ‘9’
WhiteSpace - any of space, tab, or new-line (carriage return/line feed)

3.2.2 Intrinsic Lexical Actions

Advance - add the transition character to the token buffer and advance the input pointer to the next character

Flush - clear the token buffer and advance the input pointer to the next character

Ignore - ignore the current character (i.e. do not add it to the token buffer) and advance the input pointer to the next character

Match - add the current character to the token buffer but do not advance the input pointer to the next character.

InitAdvance - works exactly as Advance but informs the scanner to register the starting line and column positions of the token

InitIgnore - analagous to InitAdvance; replace the word “Advance” with “Ignore”

InitMatch - analagous to InitAdvance; replace the word “Advance” with “Match”

PushBack - push back the current character. Note: when one uses this routine, he or she should return an attribute, as an infinite loop in the scanner is possible if the destination state is the same as the source state.

PushBackTwo - push back the current character and the previous character placed in the token buffer. The note attached to PushBack applies to PushBackTwo with one addition: when PushBackTwo is used, it is important (to prevent lexical errors) that the token buffer contains a positive number of characters.

PushBackThree - analagous to PushBackTwo; replace “previous character” with “two previous characters” and “a positive number of characters” with “two or more characters”.

3.3 DFA Considerations

In encoding a DFA it is particularly important that the language implementor be concerned with two aspects:

- every state in the DFA must be reachable

- every state in the DFA must lead to an accepting state

While neither of these two conditions assures an individual of the correctness of a lexical analyzer or the language it accepts, it minimizes the likelihood of a scanner being incorrect or exhibiting worst case behavior (like entering into an infinite loop).

The Apt Scanning Tool will inform the user of the existence of either of the two conditions above. The reachability test is determined via a transitive closure operation applied to the DFA from the start state. As mentioned in the discussion of syntax, the start state is determined from the first transition rule. The test of whether all states lead to an accepting state is also determined from a transitive closure operation.

It is important for the reader to understand why the two conditions are important. The first condition, if not satisfied, is a potentially serious problem. A set of states which is expected to perform the analysis of a lexical element in fact does nothing, because it corresponds to a piece of dead code. The second condition, if not satisfied, is usually a very serious problem. If a state is entered which cannot lead to an accepting state, an infinite loop is certain to occur. Given the input to the actual scanner, it might be the case that the state would never be entered; the scanner would not be correct nonetheless. It is possible for the scanning tool to determine a possible string which causes the scanner to get into a loop (though we currently are not supporting this). In any case, we urge the reader to rectify the problem with the DFSM in the event one of the above conditions is diagnosed to be present in his or her DFSM.

3.4 Command Line Interface

Analagous to the Apt Parsing Tool (APT), the interface to the Apt Scanning Tool is command line oriented. The figure below illustrates the screen of text which appears when you invoke the Apt Scanning Tool with no command line arguments:

Apt Scanning Tool 3.0
Copyright (c) 1991 -- Apt Technologies

Apt Scanning Tool -- ANSI-C Language Edition

An AST command line uses the following syntax:

% ast-c <file-name> <options>

where:
<file-name> ::= <unix-path>
<options> ::= { <option> }
<option> ::= -a (Analyze FSA only)
<option> ::= -i <list-of-header-files> (Use alternative token definition files)
<option> ::= -t (Suppress generation of token or terminal symbol enumeration)

The Apt Scanning Tool accepts as input a DFA specification file which conforms to the aforementioned syntax. Each of the options is explained below:

-a : analyze FSA only

Option a tells the Apt Scanning Tool to merely perform the analyses of reachability and states which can reach accepting states. If either of the analyses fails, a report of unreachable states and states which do not lead to accepting states is produced. Code generation is completely suppressed. By default this option is disabled, but analysis of the FSA always occurs and cannot be disabled.

-i : include header files

Option i informs AST to generate includes of all listed header files in the output code files. This option is commonly used to provide alternative defintions of the terminal symbol values.

3.5 Output files

Two files are produced by the Apt Scanning Tool when code generation is not suppressed: the code file and the export file. Assume the input specification file is named scanner.scn. Then the following files will be produced:

- language.c

This is the code file. It contains a complete scanner and is directly encoded from the specification file. The scanner is effectively a big switch statement in the target language with a number of local declarations to mimic the behavior of the DFSM.

- language.h

This is the export file. It contains an interface to the scanner, as well as two enumeration types (which are used by the scanner and required for a parser): TerminalDefitions (which can be read as LexicalDefinitions or TokenTypes) and States. The terminal definition list is derived from the attributes used in the specification file and are preceded by “T_”. The states are derived from the specification file (both the source states and destination states) and are preceded by “S_”. The use of enumeration types enhances the readability of the code file (language.c) and the ability to integrate the scanner with the parser (produced by APT for example).

4.0 Apt Node Tool

4.1 Introduction and Overview and Features

The Apt Node Tool is a tool designed to generate code for the allocation, updation, shallow replication, deep replication, and skeletal traversal of recursive data structures.

Each data structure is specified as a node with a type system based on that found in the C language. A node structure is for all practical purposes a “union” or “variant record” structure found in the familiar programming languages C and Modula-2 (or Pascal), respectively. The fields of the node structure are either structured or scalar types.

In the design of language translators and compilers it is important to be able to rapidly build data structures during the semantic analysis phase. Almost all semantic activity centers around the construction of data structures and the management of the symbol table. With the Apt Parsing Tool, the Apt Node Tool, and the Apt Data Types, one can literally design the entire semantic phase for a language translator and write little or no code at all. Often the only code to write is to call a function to enter or to find a definition in a symbol table.

In the design of a code generator for a compiler there are two applications for the Apt Node Tool. The parse tree (or annotated parse tree) must usually be rewritten into a form which is acceptable for code generation. With the Apt Node Tool one can generate procedures which will update or replicate a node in situ. As an example of the meaning of the latter, let us think about a node named Expression. This particular node has a member structure named BinaryExpression, which is defined as an operator with two subexpressions (each of which is a node of type Expression). Assume we have some other structures which are also lumped into node Expression which are conversion operators. Suppose we have a binary expression with an addition operator and two subexpressions of types integer and real. We need to convert the integer subexpression into a real subexpression. The following pseudocode describes how one can achieve the conversion with the aid of functions which can be generated with the node tool.

- replicate the integer subexpression eL as eL’
- change the type of the eL into a conversion expression
- establish the subexpression of the conversion expression as eL’

The above example can be achieved in two steps with procedures generated by the Apt Node Tool. Trees can be rewritten with ANT procedures without the trees being passed by reference.

4.2 Specification File Syntax

```
File -> Items
Items -> { Item }
Item -> Enum | Include | Node
Enum -> enum ident { Tags }
Include -> include fileIdent
Node -> node ident { Elements }
Tags -> Tag { , Tag }
Tag -> identifier
Elements -> { Element }
Element -> Slot | Decl
Slot -> slot ident { Decls } _Slot SemiOpt
Decls -> { Decl }
Decl -> NullOp Specs Vars ;
NullOp -> [ null ]
```

Specs -> SpecList
SpecList -> ClassSpec
SpecList -> { Spec }
ClassSpec -> enum ident
ClassSpec -> ident
ClassSpec -> struct ident
ClassSpec -> union ident
Spec -> TypeSpec | StClSpec | TypeQual
TypeSpec -> void | char | int | float | double | short | long | signed | unsigned
StClSpec -> auto | static | extern
TypeQual -> const | volatile
Vars -> Var { , Var }
Var -> Ptrs ident Arrays
Ptrs -> { * }
Arrays -> { Array }
Array -> { [[integer]] }

As the specification file syntax includes much syntax for the declaration of C data types, it has a complicated appearance. Actually, the syntax is easy to comprehend by looking at the first rules and then working downward, since the grammar is arranged hierarchically.

A specification file is a list of items. An item is either an include, a node, or an enumeration. The include rule permits one to include other definitions (such as other data types and enumerations) for the definition of nodes. An enumeration facility is provided, so members of a given node can be declared as an instance of an enumeration type. The enumeration type, as defined in ANT, differs from C in that no initial value can be specified. Thus, an enumeration is defined in ANT to always start with the value of zero. The node item, the most important item, is the item which allows the flexible specification of user data structures.

A node declaration looks much like a union declaration in the C programming language, except for the keyword and the need to enclose it in a structure declaration with a tag member associated with the union. To declare a node, simply specify the node name (which is not optional) and a list of members. The members may be any C data type (either a scalar or structured type).

As mentioned above, the members of a node are data types found in the C programming language. A member of a node can be optionally qualified as null, if the user desires the member variable(s) to be invisible. In other words the member variable, while present in the generated code for the node, is never generated as a parameter to any of the functions which allocate or manipulate an instance of the node type.

In the final section of the manual we will illustrate a complete example translator which uses the three tools to solve a problem. ANT will be used to build the data structures required for code generation.

Analagous to the Apt Scanning Tool (AST), the interface to the Apt Node Tool is command line oriented. The figure below illustrates the screen of text which appears when you invoke the Apt Node Tool with no command line arguments:

Apt Node Tool 3.0
Copyright (c) 1991 -- Apt Technologies

Apt Node Tool

ANT accepts a node description file and translates it into a header file and a code file. The header file contains enumeration types, structure types, and union types, while the code file contains routines to allocate the different structures.

To put the ANT to work, supply files of the form *.nt at the command line prompt:

% ant file1.ant file2.ant ... filen.ant

The command line syntax merely requires the user to specify one or more files (containing node descriptions) to be translated. Currently, we have no command line flags to suppress the generation of code which is redundant for a specific application.

4.3 Output Files

Two files are produced by the Apt Node Tool: a code file and a definition/export file. The definition/export file must be included by the application which needs to build/ maintain nodes to ensure proper usage of the functions and linkage. Assume the specification file is named nodes.ant

- nodes.c

For each node (and structure within the node) functions to allocate (New), replicate (Duplicate), and modify (Update) can be found in the file. In a future release two types of node duplication (shallow and deep) will be supported by ANT, as well as a "skeleton" traversal routine for recursive descent into a (recursively defined) data structure.

- nodes.h

Every node is translated into a structure type. The structure contains a tag field and a union. The tag field is of an enumeration type (which is named after the node); the union field is of a union type. The enumeration type, the union type, and the structure type are all defined types in the file which can be used by an application which includes the file. The enumeration type provides a list of tags which can be used to label the node (which are named after the node and the structure member). We omit the specific details of the code generated here, because applications really need not concern themselves with the implementation details of the data structures. An application can assume it is has the appropriate functionality to create an instance of a certain node and be complacent with the idea of it being done properly.

Now that we have a general understanding of how the various tools work, we proceed toward a discussion of the pragmatics of how the various tools are used to solve a particular translation problem. From the example it is our hope the reader will be amply prepared to embark on a translation problem of greater complexity with the assistance of the Apt Compiler Toolkit.

5.0 Apt Compiler Toolkit Libraries

5.1 The ADT Library

5.1.1 ADTs in General

ADT library basically provides the data types and appropriate functions to create, manipulate, and dispose instances of these data types. Each ADT module contains the data type definition, and related function templates and their implementations. Basic data types, like queue, stack using queue, stack using array, avl tree, bidirectional queue (called dequeue), symbol table using avl tree, symbol table using hash table are supported by the ADT modules. Each data type is defined and made available to the public use through the header files.

Most of the ADTs have common operations defined through the usage of the function templates. These operations are grouped into function classes that include generation, disposal, service, access, helper and application functions.

ADTs support the storage, access and manipulation of generic data types within themselves. Hence, for example, character strings, integers, real numbers, etc. can be stored and manipulated in ADTs generically.

In the following sections each ADT implementation is described in detail covering the definition of the ADT, function templates for the operations, and the usage of the parameters in these templates.

5.1.2 Common Types and Functions for ADTs

- Comparison Function Type

These function types are declared and passed to the ADT functions that perform some lookup operation to find a specific data item over the whole data structure. The data items in the data structure are compared with the searched data item specifications according to the parameters specified in the compare function parameter template. The comparison function dectates how the actual comparison is to be performed between the data items in the structure and the searched data item. These comparison functions are necessary to support the generic data item handling in the ADTs.

The type definition for the comparison functions is declared in APT.H module, and the format is as follows:

typedef int (*ComparisonFunction)(void*, void*).

In this template, two void pointers correspond to two data items that are to be compared according to the actual implementation of the comparison function.

The function return code indicates the result of the comparison.

- Disposal Function Type

These function types are declared and passed to the ADT functions that perform some delete operation to delete a specific data item from the actual data structure. The dispose function dectates how to de-allocate the memory space reserved for the data item to be deleted. The disposal function type specifications are the result of the support for generic data item hadling in the ADTs.

The type definition for the disposal functions is declared in APT.H module, and the format is as follows:

typedef void (*DisposeFunction)(void*).

Here, the void pointer is the pointer that points to the data item to be deleted. The internal structure of the data item is de-allocated according to the actual implementation of the dispose function.

- Constructor Function

Constructor function is basically used to allocate a new instance of an ADT. Each ADT module has a specific constructor function, associated with the name of the ADT.

Although there are some ADTs using different templates, the general template of a constructor function is as follows:

<ADT_Type_Pointer> <ADT_Name>New(void);

Constructor function templates are declared in <ADT_Name>.H files. Each constructor function call returns a pointer to a new instance of the corresponding ADT.

- Destructor Function

Destructor function is basically used to de-allocate an instance of an ADT. Each ADT module has a specific destructor function, associated with the name of the ADT. The internal de-allocation of the data structure is performed according to the disposal function pointer specified in the destructor function call.

Although there are some ADTs using different templates, the general template of a destructor function is as follows:

void <ADT_Name>Dispose(<ADT_Pointer>, DisposeFunction);

Destructor function templates are declared in <ADT_Name>.H files.

5.1.3 ADT Queue

- Definitions

Generic Queue Item Definition

```
typedef struct _QueueItem {
      void *element;
      int type;
      struct _QueueItem *next;
} _QueueItem, *QueueItem;
```

Queue Definition

```
typedef struct _Queue {
      struct _QueueItem *head;
      struct _QueueItem *tail;
      int size;
} _Queue, *Queue;
```

- Exported Function Prototypes

Service Functions

void QueueDispose(Queue q, DisposeFunction f);

Dispose the whole queue q using the disposal function f to de-allocate the items of q.

void *QueueFindandRemove(Queue q, void *v, ComparisonFunction f);

Similar to QueueFind, but, if the item is found, it is removed from the queue q and a pointer to the removed queue item is returned.

void *QueueFindTypeandRemove(Queue q, int typeToFind);

Similar to QueueFindType, but, if the data item is found, it is removed from the queue q and a pointer to the removed queue item is returned.

void *QueueGet(Queue q);

Return the element field of the item pointed by the head of pointer of queue q. The item pointed by the head pointer is removed from the queue.

Queue QueueNew(void);

Allocate a new instance of ADT Queue and return the pointer to the allocated queue.

void QueuePut (Queue q, void *item, int type);

Place the data item item at the end of the queue (i.e. Tail pointer of the queue points to the newly put item after the operation is completed). The type field of the new queue node is set to type.

void QueuePutOnPriority(Queue q, void *item, int type, ComparisonFunction f);

Place the data item item at the end of the items in the priority group satisfying the function f. The type field of the new queue node is set to type.

int QueueSize(Queue q);

Return the number of items in the queue q.

Access Functions

void *QueueFindType(Queue q, int typeToFind);

Find and return the first queue item with type typeToFind from the queue q. The data item is not removed from the queue.

void *QueueItemElement(QueueItem item);

Return the data element part of the queue item item.

int QueueItemType(QueueItem item);

Return the type part of the queue item item.

void *QueueLook(Queue q);

Return the element field of the item pointed by the head pointer of queue q. No modification is done on the queue q.

void *QueueFind(Queue q, void *itemToFind, ComparisonFunction f);

Find and return the queue item from the queue q. The comparison for search is done using the function f. The queue is searched for the data item specified by the void pointer itemToFind. The data item is not removed from the ADT Queue.

Helper Functions

QueueItem QueueNext(QueueItem item);

Return the queue item pointed by the next field of the queue item item.

QuueItem QueueSeek(Queue q, int offset);

Return the queue item coming after offset redirections from the head of the queue.

QueueItem QueueHead(Queue q);

Return the item pointed by the head pointer of queue q.

QueueItem QueueTail(Queue q);

Return the item pointed by the tail pointer of queue q.

Application Functions

void QueueApply(Queue q, ApplyFunction f);

Apply the function f to queue q.

void QueueApply1(Queue q, void *v, ApplyFunction1 f);

Apply the function f to the whole queue q, keeping the result of the application in the variable pointed by void pointer v.

void QueueApply2(Queue q, void *v1, void *v2, ApplyFunction2 f);

Apply the function f to the whole queue q, keeping the results of the application in the variables pointed by two void pointers v1 and v2.

void QueueApply3(Queue q, void *v1, void *v2, void *v3, ApplyFunction3 f);

Apply the function f to the whole queue q, keeping the results of the application in the variables pointed by three void pointers v1, v2 and v3.

5.1.4 ADT Deque

- Definitions

Deque Link Types

```
typedef enum _DequeLinkTypes {
    NoLink = -1, Top, Bottom, NumberOfLinks
} DequeLinkTypes;
```

Deque Item Definition

```
typedef struct _DequeItem {
    void *element;
    int type;
    struct _DequeItem *next[NumberOfLinks];
} _DequeItem, *DequeItem;
```

Generic Deque Definition

```
typedef struct _Deque {
    struct _DequeItem *link[NumberOfLinks];
    int size;
}
```

- Exported Function Prototypes

Service Functions

Deque DequeNew();

Allocate a new instance of ADT Deque and return the pointer to the newly allocated deque.

void DequeDispose(Deque deq, DisposeFunction d);

Dispose deque deq using dispose function d to deallocate the nodes of deque.

void DequePut(Deque deq, void *element, int type, DequeLinkTypes l);

Allocate a new deque node, set the element field to element and type field to type. Put the new node on side l of deque deq.

void DequePutOnPriority(Deque deq, void *element, int type, DequeLinkTypes l, ComparisonFunction f);

Allocate a new deque node, set the element field to element and type field to type. Put the new node on side l of the node group satisfying the comparison function f.

void *DequeGet(Deque deq, DequeLinkTypes l);

Remove the deque node on side l of deque deq. Return the pointer to the element field of the removed node.

Access Functions

void *DequeFind(Deque deq, void *element, DequeLinkTypes l, ComparisonFunction f);

Return the pointer to the element field of the deque node which satisfies the comparison function f when applied to itself and the searched element specification. The search is done starting from side l of deque deq.

int DequeSize(Deque deq);

Return the number of nodes in deque deq.

Helper Functions

void *DequeItemElement(DequeItem d);

Return the pointer to the element field of deque item d.

int DequeItemType(DequeItem d);

Return the type of deque item d.

DequeItem DequeNext(DequeItem d, DequeLinkTypes l);

Return the pointer to the deque node which is on side l of deque node d.

DequeItem DequeSide(Deque deq, DequeLinkTypes l);

Return the pointer to the deque node which is on side l of deque deq.

Application Functions

void DequeApply(Deque deq, DequeLinkTypes l, ApplyFunction f);

Apply function f to deque deq, starting on side l.

5.1.5 ADT AStack

- Definitions

Stack Operation Error Codes

```
typedef enum _AStackErrorTypes {
    AStackOK,
    AStackOverflow,
    AStackUnderflow,
    AStackRangeError,
    AStackTypeMismatch,
    AStackBadDisplacement
} AStackErrorTypes;
```

Static Stack Item Types

```
typedef enum _AStackItemTypes {
    NoType, Byte, LongWord, Pointer, Word
} AStackItemTypes;
```

Static Stack Item Definition

```
typedef struct _AStackItem {
    enum _AStackItemTypes type;
    union {
        unsigned char b;
        unsigned short w;
        unsigned long lw;
        void *ptr;
    } u;
} _AStackItem, *AStackItem;
```

Static Stack Definition

```
typedef struct _AStack {
    int size;
    int max;
    enum _AStackErrorTypes error;
    struct _AStackItem *stack;
} _AStack, *AStack;
```

- Exported Function Prototypes

Service Functions

AStack AStackNew(int i);

Allocate a new instance of ADT AStack of static size i and return the pointer to the allocated static stack.

void AStackDispose(AStack s, DisposeFunction f);

Dispose the static stack s using the disposal function f to de-allocate the items of s.

unsigned char AStackGetByte(AStack s, int disp);

Return the data field of the static stack node relative to disp nodes to the size of the stack. The stack error code is set appropriately. If the data type of the referred node is not byte, error code reflects a type mismatch (i.e. set to AStackTypeMismatch). If the displacement disp is not within the allowable displacement range (determined according to the size of the stack), error code is set to AStackBadDisplacement. A successful operation sets the error code to AStackOK. Successful operation results in the removal of the corresponding stack node, and the data field of the removed node is returned.

unsigned long AStackGetLongWord(AStack s, int disp);

Similar to AStackGetByte, replacing byte by long instead.

void *AStackGetPointer(AStack s, int disp);

Similar to AStackGetByte, replacing byte by pointer. Return the pointer field of the removed stack node.

unsigned short AStackGetWord(AStack s, int disp);

Similar to AStackGetByte, replacing byte by word.

int AStackSize(AStack s);

Return the number of items currently in the stack s.

unsigned char AStackPopByte(AStack s);

If the stack s is not empty and the data type of the top node is equal to byte, pop the top node from the stack s and return the data field of the removed node. Set the error code to indicate type mismatch if the top of the stack is not of type byte. Set the error code to AStackUnderflow if the stack is empty before the operation. Successful operation sets the error code to AStackOK.

unsigned long AStackPopLongWord(AStack s);

Similar to AStackPopByte, replace byte with longword.

void *AStackPopPointer(AStack s);

Similar to AStackPopByte, replace byte with pointer.

unsigned short AStackPopWord(AStack s);

Similar to AStackPopByte, replace byte with word.

void AStackPushByte(AStack s, unsigned char c);

Push a new stack node containing the byte c on top of the stack s. Set the error code appropriately, according to the size of the stack (i.e. AStackOK or AStackOverflow).

void AStackPushLongWord(AStack s, unsigned long l);

Similar to AStackPushByte, but instead push long l.

void AStackPushPointer(AStack s, void *p);

Similar to AStackPushByte, but instead push pointer p.

void AStackPushWord(AStack s, unsigned short w);

Similar to AStackPushByte, but instead push word w.

void AStackSpill(AStack s, int i, DisposeFunction f);

Pop and discard the top i nodes of the stack s, using function f to de-allocate the node information. The stack error code is set according to the size of the stack and i.

Access Functions

unsigned char AStackLookByte(AStack s);

If the stack s is not empty and the type of the top stack node is byte, return the data field of the node and set the error code to AStackOK. Otherwise if the stack is empty, set the error code to AStackUnderflow. If the type of the top node is not byte, set the error code to AStackTypeMismatch.

unsigned long AStackLookLongWord(AStack s);

Similar to AStackLookByte, replace byte with long word instead.

void *AStackLookPointer(AStack s);

Similar to AStackLookByte, replace byte with pointer instead.

unsigned short AStackLookWord(AStack s);

Similar to AStackLookByte, replace byte with word instead.

AStackItemTypes AStackLookType(AStack s);

Return the data type of the top stack node of stack s.

AStackErrorTypes AStackError(AStack s);

Return the error code field of stack s.

Helper Functions

int AStackCheckByte(AStack s, int disp);

Check if the type of the disp’th node from the top of the stack s is byte or not.

int AStackCheckLongWord(AStack s, int disp);

Check if the type of the disp’th node from the top of the stack s is longword or not.

int AStackCheckPointer(AStack s, int disp);

Check if the type of the disp’th node from the top of the stack s is pointer or not.

int AStackCheckWord(AStack s, int disp);

Check if the type of the disp’th node from the top of the stack s is word or not.

AStackItemTypes AStackCheckType(AStack s, int disp;

Return the type of the disp’th node from the top of the stack s.

int AStackEmpty(AStack s);

Return if the stack s is empty or not.

int AStackNotEmpty(AStack s);

Return if there are some items in stack s or not.

void AStackClearError(AStack s);

Set the error code of stack s to AStackOK.

5.1.6 ADT Stack

- Definitions

Generic Stack Item Definition

```
typedef struct _StackItem {
    void *element;
    int type;
    struct _StackItem *next;
} _StackItem, *StackItem;
```

Dynamic Stack Definition

```
typedef struct _Stack {
    struct _StackItem *head;
```

```
    struct _StackItem *tail;
    int size;
} _Stack, *Stack;
```

- Exported Function Prototypes

Service Functions

void StackDispose(Stack s, DisposeFunction f);

Dispose the stack s using dispose function f to de-allocate the internal stack node information.

Stack StackNew(void);

Allocate a new instance of ADT dynamic stack and return the pointer to the allocated stack.

void *StackPop(Stack s);

Remove the top node from the stack s and return the pointer to the element field of the removed stack node.

void StackPush(Stack s, void *n, int type);

Push a new node onto stack s. The type of the new node is set to type, and the element field of the new node is set to pointer n.

void StackPushOnPriority(Stack s, void *n, int type, ComparisonFunction f);

Push a new node on top of the nodes forming a priority group in stack s according to comparison function f. The type of the new node is set to type, and the element field of the new node is set to pointer n.

int StackSize(Stack s);

Return the number of nodes in stack s.

Access Functions

void *StackFind(Stack s, void *item, ComparisonFunction f);

Find and return the stack node element from the stack s. The comparison for search is done using the comparison function f. The stack is searched for the data item specified by the void pointer item. The stack node is not removed from the stack s.

StackItem StackHead(Stack s);

Return the pointer to the head stack node of stack s.

StackItem StackTail(Stack s);

Return the pointer to the tail stack node of stack s.

void *StackItemElement(StackItem item);

Return the pointer to the element field of the stack node item.

int StackItemType(StackItem item);

Return the type field of the stack node item.

void *StackLook(Stack s);

Return the pointer to the element field of the top stack node of stack s. The stack node is not removed from the stack.

Helper Functions

StackItem StackNext(StackItem item);

Return the pointer to the next stack node which is pointed by the next field of stack item item.

Application Functions

void StackApply(Stack s, ApplyFunction f);

Apply function f to every node in the stack s.

5.1.7 ADT AVLTree

- Definitions

Balance Type Definition

```
typedef enum _AVLTreeBalanceTypes {
    LEFTHIGH, EQUALHIGH, RIGHTHIGH
} AVLTreeBalanceTypes;
```

AVLTree Item Definition

```
typedef struct _AVLTreeItem {
    AVLTreeBalanceTypes balance;
    int type;
    void *element;
    struct _AVLTreeItem *left;
```

```
    struct _AVLTreeItem *right;
} _AVLTreeItem, *AVLTreeItem;
```

AVLTree Definition

```
typedef struct _AVLTree {
    int size;
    struct _AVLTreeItem *root;
} _AVLTree, *AVLTree;
```

- Exported Function Prototypes

Service Functions

AVLTree AVLTreeNew(void);

Allocate a new instance of ADT AVLTreeand return the pointer to the allocated tree.

void AVLTreeDispose(AVLTree t, DisposeFunction f);

Dispose the AVLTree t using the disposal function f to de-allocate the nodes of the tree.

int AVLTreeInsert(AVLTree t, void *n, int type, ComparisonFunction f);

Insert a new tree node with the element pointer field n and type type. Use comparison function f to search for the already existance of the new node.

int AVLTreeDelete(AVLTree t, void *n, ComparisonFunction c, DisposeFunction d);

Delete the tree node which satisfies the comparison function c, which is applied to the tree node and the element information specified by n. The tree node is de-allocated using the disposal function d.

void *AVLTreeDeleteLeftMost(AVLTree t);

Remove the left most tree node of AVLTree t and return the pointer to the element field of the removed tree node.

void *AVLTreeDeleteRightMost(AVLTree t);

Remove the right most tree node of AVLTree t and return the pointer to the element field of the removed tree node.

Access Functions

void *AVLTreeFind(AVLTree t, void *n, ComparisonFunction f);

Find the AVLTree node which satisfies the comparison function f. The comparison function is applied to tree node and n during the search. Return the pointer to the element field of the found tree node.

void *AVLTreeFindLeftMost(AVLTree t);

Return the pointer to the element field of the left most node of AVLTree t.

void *AVLTreeFindRightMost(AVLTree t);

Return the pointer to the element field of the rigth most node of AVLTree t.

int AVLTreeHeight(AVLTree t);

Return the level of the AVLTree t.

AVLTreeBalanceTypes AVLTreeItemBalance(AVLTreeItem item);

Return the balance field of the AVLTree node item.

void *AVLTreeItemElement(AVLTreeItem item);

Return the pointer to the element field of AVLTree node item.

int AVLTreeItemType(AVLTreeItem item);

Return the type of the AVLTree node item.

int AVLTreeLeftHeight(AVLTree t);

Return the left level of the AVLTree t.

int AVLTreeRightHeight(AVLTree t);

Return the right level of the AVLTree t.

int AVLTreeSize(AVLTree t);

Return the number of nodes in AVLTree t.

AVLTreeItem AVLTreeRoot(AVLTree t);

Return the pointer to the root node of AVLTree t.

Helper Functions

AVLTreeItem AVLTreeItemLeft(AVLTreeItem item);

Return the pointer to the left child of AVLTree node item.

AVLTreeItem AVLTreeItemRight(AVLTreeItem item);

Return the pointer to the right child of AVLTree node item.

Application Functions

void AVLTreeInorderApply(AVLTree t, ApplyFunction f);

Apply function f to the AVLTree t, using inorder traversal of the tree.

void AVLTreePostorderApply(AVLTree t, ApplyFunction f);

Apply function f to the AVLTree t, using postorder traversal of the tree.

void AVLTreePreorderApply(AVLTree t, ApplyFunction f);

Apply function f to the AVLTree t, using preorder traversal of the tree.

5.1.8 ADT Buffer

- Definitions

Buffer Definition

```
typedef struct _Buffer {
      Deque data;
      int pos;
      char *text;
} _Buffer, *Buffer;
```

- Exported Function Prototypes

Service Functions

Buffer BufferNew();
Allocate a new instance of ADT Buffer and return the pointer to the allocated buffer.

void BufferDispose(Buffer b);
Dispose buffer b, freeing the reserved space for b.

char BufferDelChar(Buffer b);
Remove the last character from buffer b and return the pointer to the removed character. If buffer b is empty before the deletion, return EOF character.

void BufferAddChar(Buffer b, char ch);
Add character ch to the end of buffer b.

void BufferAddString(Buffer b; char *str);
Add character string that is pointed by str to the end of buffer b.

Access Functions

int BufferSize(Buffer b);

Return the number of characters in buffer b.

Application Functions

char *BufferToString(Buffer b);
Form a string of characters that are in buffer b and return the pointer to the resulting string.

- Server ADT's

ADT Deque

5.1.9 ADT BufIO

The current version of BufIO implements only buffered input operations from a file. Output operations will be supported in the coming versions of ADT library.

- Exported Function Prototypes

ServiceFunctions

int BufIOGetChar(FILE *file);

Make a buffered character input from the file file. If the buffer is empty, the character is read from the file. Otherwise, the first character in the buffer is removed from the buffer and returned.

char *BufIOGetString(FILE *file);

Make a buffered character string input from the file file.

void BufIOUnGetChar(int ch, FILE *file);

Unget the character ch to the beginning of the buffer associated with file file.

void BufIOInitialize(void);

Initiate the buffered input of the characters by allocating a buffer structure.

Access Functions

int BufIOBufferSize(FILE *file);

Return the number of characters in the input buffer associated with file file.

5.1.10 ADT Hash Table

- Definitions

Hash Table Element Definition

```
typedef struct _HashTableElement {
    void *key;
    int tag;
    int type;
    union {
    void *dataPtr;
    QueuePtr qPtr;
    } u;
} HashTableElement, *HashTableElementPtr;
```

Hash Table Definition

```
typedef struct _HashTableAVL {
    int size;
    HashFunctionPtr hash;
    HashCompareFunctionPtr compare;
    HashTableElementDisposeFunctionPtr elementDisposeFunction;
    AVLTreePtr *bucket;
    HashTableAVLPtr scopeLink;
} HashTableAVL;
```

- Exported Function Prototypes

Service Functions

Access Functions

Helper Functions

Application Functions

- Server ADT’s

ADT Queue, ADT AVL Tree.

5.1.11 ADT AVL Table

- Definitions

AVL Table Item

```
typedef struct _AVLTableItem {
    char *key;
```

```
    void *element;
    int type;
} _AVLTableItem, *AVLTableItem;
```

AVL Table

```
typedef struct _AVLTable {
    struct _AVLTable *ScopeLink;
    AVLTree Space;
} _AVLTable, *AVLTable;
```

- Exported Function Prototypes

Service Functions

AVLTable AVLTableNew(AVLTable scopeLink);
Allocate a new instance of ADT AVLTable, set the scope link field of the new table to scopeLink and return the pointer to the newly allocated AVLTable.

void AVLTablePut(AVLTable table, char *key, void *element, int type);
Put an association for the (key, element, type) tuple in AVLTable table.

void *AVLTableGet(AVLTable table, char *key, int type);
Find the first association having (key, type) in AVLTable table. If found, remove the association from the table and return the pointer to the element field of the corresponding association. Otherwise return NULL.

Queue AVLTableGetAll(AVLTable table, char *key);
Find all associations having (key) in AVLTable table, remove them from the table forming a queue of (element, type) tuples of those associations. Return the queue.

Access Functions

void *AVLTableFind(AVLTable table, char *k, int type);
Find the first association having (key, type) in AVLTable table. If found, return the pointer to the element field of the corresponding association. Otherwise return NULL.

Queue AVLTableFindAll(AVLTable table, char *key);
Find all associations having (key) in AVLTable table, form a queue of (element, type) tuples of those associations. Return the queue.

AVLTable AVLTableScopeLink(AVLTable table);
Return the scopeLink field of AVLTable table.

- Server ADT’s

ADT AVL Tree, ADT Queue.

The ADT Iterators Library

5.2 The ADT Iterators Library

5.2.1 Iterators in General

An iterator for an abstract data type is an abstraction for iteration on the data type. What an iterator allows one to do depends upon the particular data structure. Iteration, to which we are accustomed, enables us to consider elements of a sequence in forward, reverse, alternating, or random fashion.

The iterator encapsulates the details of iteration for iteration patterns common to a particular data type. For most data types iteration is straightforward. A queue is typically iterated from its head to its tail, which a tree is typically iterated via on of the common traversals. In the case of a tree the traversal pattern must be done recursively, but this can be somewhat cumbersome. While we do not intend to belabor the point, suppose we had a tree which contains 1,000 words. We want to print out the first 50 words, or every one of the first 100 words, or perhaps we have some other novel iteration pattern. To achieve a recursive solution to the suggested iterations would require somewhat nontrivial recursion. With an iterator which permits the different iterations the process is suddenly simple:

1. create a tree iterator - inorder
2. count <- 1
2. advance to the first position in the iterator
3. while not at the end of the iteration and count <= 50
3.a. print the item
3.b. count <- count + 1
3.c. advance to position count + 1

The iterator abstacts the problem of iteration on an ADT and does not require the user to be familiar with the low-level routines for iteration which are provided by the ADT. Furthermore, iteration on any ADT is virtually the same, though the user must be aware of potential inefficiency which results from naive use of the iterator. As an example, backward iteration on a queue is guaranteed to be slow, even though it is achieved by the iterator routines in a manner identical to that of forward iteration.

5.2.2 ADT Queue Iterator

- Discussion

- Definitions

```
typedef struct _QueueIterator {
    int position;
    Queue queue;
    QueueItem currentItem, previousItem;
} _QueueIterator, *QueueIterator;
```

- Constructor

QueueIterator QueueIteratorNew(Queue queue, int position)

Allocate a queue iterator and set the position to point to the first addressible position in the queue. Set currentItem to point to the head of the queue and previousItem to NULL.

- Destructor

void QueueIteratorDispose(QueueIterator qi);

Deallocate the queue iterator.

- Status Functions

int QueueIteratorAtTop(QueueIterator qi);

Returns TRUE when the queue iterator, qi, points to the first addressible position in the queue; FALSE otherwise.

int QueueIteratorAtBottom(QueueIterator qi);

Returns TRUE when the queue iterator, qi, points to the last addressible position in the queue; FALSE otherwise.

int QueueIteratorAtPosition(QueueIterator qi, int position);

Returns TRUE when the queue iterator, qi, points to the position whose address is position; FALSE otherwise or when position is not valid.

- Access Functions

int QueueIteratorPosition(QueueIterator qi);

Returns the integer position of the queue iterator with respect to the queue over which it iterates. The value of position is always non-negative.

int QueueIteratorCurrentData(QueueIterator qi);

Returns the data contained in the current queue item (NULL if the current item is NULL).

int QueueIteratorPreviousData(QueueIterator qi);

Returns the data contained in the previous queue item (NULL if the previous item is NULL).

- Iteration Functions

void QueueIteratorAdvance(QueueIterator qi);

Advance to the next addressible position in the queue, if the addressible position is not the last addressible position; otherwise, remain at the last addressible position.

void QueueIteratorBackup(QueueIterator qi);

Move to the previous addressible position in the queue, if the previous position is not less than the lowest addressible position; otherwise, remain at the lowest addressible position. This function is translated into QueueIteratorRelativeSeek(qi,-1).

void QueueIteratorAbsoluteSeek(QueueIterator qi, int delta)

void QueueIteratorRelativeSeek(QueueIterator qi, int position)

5.3 The APT Library - a library specific to APT

5.3.1 Overview

The APT Library is a collection of functions which are specifically needed by projects to utilize the tables produced by the Apt Parsing Tool. While the Apt Compiler Toolkit provides a canned main module which must be linked with the generated code modules and user-specified semantics (and possibly other modules), one ought to know about the different functions available. The functions available include initialization functions, LL(1) parsers, EOPM parsers, and miscellaneous functions.

5.3.2 Functions

- Initialization and Termination Functions

void ParseInitialize();

Allocate the data structures required for any of the LL(1) Parsers

void ParseTerminate();

Free all data structures allocated by ParseInitialize.

- LL(1) Parsers

Parse(FILE *file, VOID_FN_TOKEN printToken);

Parse the token stream generated from file (which must be opened for read prior to the call). Execute semantics as phrases are determined. A function which prints tokens, printToken, may be supplied to print a token every time the LL(1) parser successfully matches a terminal symbol. Supply NULL, if token printing is not desired.

ParseEOPMOut(FILE *file, FILE *eopmFile, VOID_FN_TOKEN printToken);

Parse the token stream generated from file into eopmFile (which must be opened for write prior to the call). No semantics are executed, as a companion function exists (see EOPM Parsers). As with Parse, a function which prints tokens may be supplied with the same regulations.

ParseEOPMOutQ(Queue queue, FILE *eopmFile, VOID_FN_TOKEN printToken);

Identical to ParseEOPMOut with one exception. A queue of tokens is passed instead of a file to be scanned for tokens. A function is provided (see Miscellaneous) which builds a token queue from a file.

- EOPM Parsers

void ParseEOPMIn(FILE *eopmFile, VOID_FN_TOKEN printToken);

The companion routine to ParseEOPMOut. Execute semantics on token stream which was successfully parsed into phrases (and written to eopmFile). As with the parse functions, a function may be supplied to print a token. Each time a token is read from eopmFile, it is printed with printToken.

- Miscellaneous Functions

void SetDebugOption(ParseDebugOptions option);

Enable one of the debugging options supported by the LL(1) parsers and EOPM parsers:

DebugToken
DebugPredictionStack
DebugSemanticStack
DebugSemanticAction

What is being debugged is self-explanatory. What output occurs requires some explanation. The output occurs in a line-oriented fashion. One line will appear for each enabled debug option. For tokens and actions, the output is simply the name of the token (preceded by the word “token”) or action (preceded by the word “action”). The prediction stack and semantic stacks will be dumped on a line with the leftmost symbol representing the top of the stack. To enable an option requires a separate call for each one. A future release will provide more elegant support for debugging.

void ResetDebugOption(ParseDebugOption);

Disable one of the debugging options (which was enabled by SetDebugOption).

Queue BuildTokenQueue(FILE *file, int end);

Build a queue of tokens which are generated via application of the scanner, Scan, to the file, file, which must be opened for input. When the token, end, is encountered, it is the last token scanned, and the queue is returned.

5.4 The AST Library - a library specific to AST

5.4.1 Overview

The AST Library is a library of functions required for the maintenance of the state of the scanner. As mentioned in the discussion of the APT Library, one need not know much about the functions of the AST Library, since a canned main module is provided.

5.4.2 Functions

ScanInfo ScanInfoNew()

Allocate an instance of ScanInfo to maintain information about the current input file.

void ScanStateInit(ScanInfo *info, FILE *file);

Set file as the current file for lexical analysis (scanning). The file must have just been opened for input, else all subsequent lexical analysis is undefined.

void ScanStateSwitch(ScanInfo *info, FILE *file);

Allocate a new instance of ScanInfo (with ScanInfoNew) and set the current input file to file. The last allocated instance is placed beneath the current instance on a stack, so it can be reverted to at some point, if desired.

void ScanStateRevert(ScanInfo *info);

Revert to the scanning state which is pointed to by the current scanning state. Free the space being occupied by the current scanning state.

6.0 How to Design a Translator

6.1 The Translation Problem

In this section we will illustrate how ACT is used to develop a translator for a reasonably interesting language. Our problem is to design an evaluator for a series of equations. Each equation is an assignment expression such that a variable is defined in terms of a function of other variables. The other variables can be defined in terms of other variables (and et cetera); however, no variable can be directly or indirectly defined in terms of itself (to keep the problem from gravitating away from a pedagogical one).

6.2 Language Definition

As is disucssed in many texts on the implementation of programming language translators and compilers, a language is defined at four levels: lexical, grammatical, semantics, and evaluation (or code generation). Each of the four levels is discussed in the following four sections. Their implementation in the framework of the Apt Compiler Toolkit is the subject of the next section.

6.2.1 EBNF Grammar

Below is a grammar for the Equation Language which was outlined in section 6.1. We define it in EBNF form to be concise. As an exercise for the reader, convert the EBNF grammar into LL(1) form. We do not require you to do this exercise but consider it worth your while from a pedagogical standpoint.

Program ::= { Equation }
Equation ::= Var := Expression
Expression ::= Term { AddOp Term }
Term ::= Factor { MulOp Factor }
Factor ::= Base { ExpOp Base }
Base ::= integer | number | Var | (Expression)
MulOp ::= * | /
AddOp ::= + / -

ExpOp ::= **
Var ::= identifier

A program is defined to be a list of equations, which possibly is empty. An equation is an assignment of an arithmetic expression to a variable. The different types of expression are addition, subtraction, multiplication, division, and exponentiation in increasing precedence. The precedence levels can be overridden via the use of parentheses (also known as the grouping operators). The operands of expressions can either be numbers, variables, or expressions (because of grouping).

In our definition of the language syntax it is especially important to note that we are not defining semantics. For instance the details of intermediate representation, translation, and evaluation are omitted. The discussion of semantics is deferred until we have discussed the lexical elements of the equation language.

6.2.2 Lexical Elements (Tokens)

The definition of lexical elements is commonly derived from two sources:

- the language grammar (above)
- non-syntactic lexical elements

The lexical elements from the language grammar include all of the terminal symbols. We caution the reader that we are making a distinction which is not commonly made in the definition of a language. The Pascal and C programming language grammars contain definitions for lexical elements which are nonterminal symbols. We believe this is a rotten idea, because it is difficult to comprehend the difference between lexical and syntactic constructs. As we only intend to caution you, we now define the lexical elements which are terminal symbols.

:=	the assignment operator
+	the addition operator
-	the subtraction operator
*	the multiplication operator
**	the exponentiation operator
/	the division operator
identifier	an alphabetic character followed by alphabetic or numeric char
number	a signed integer or floating point number
(	left parenthesis
)	right parenthesis

Also included among the lexical elements are non-syntactic constructs. A non-syntactic construct is one which has no syntactic value. One non-syntactic construct is the comment, which is found in many useful programming languages. There are others, but we will not discuss them here.

We define two styles of comment: line-comments and free-comments. Line-comments are defined as a pound sign, #, followed by any characters and terminated by end-of-line. Free-comments are defined to be identical to C comments: an opener /*, followed by any characters, and terminated by a closer */. As in the C language, comments may not be nested.

6.2.3 Intermediate Representation

There is one type of structure to be built based on syntax alone: an expression. An expression is either a binary expression, or a leaf. A binary expression is defined to be a binary operator (one of the above addition, multiplication, or exponentiation operators). A leaf expression is defined to be either a number or an identifier.

In addition to the above data structure, a symbol table will be used to maintain associations between a variable and an expression (which is really the definition of an equation). A list of variables to be evaluated will also be maintained in a queue. We will encode semantic actions for these purposes; however, most semantic actions will be generated by the Apt Node Tool.

Now we will encode APT, AST, and ANT specification files for the language grammar, then lexical scanner, and the data structures.

6.2.4 Evaluation Semantics

Once the intermediate representation has been determined, we can define the semantics of the equation language. We know from syntax that a program is a list of equations. As we encounter an equation, we can maintain a list of variables which appear on the left hand side of the assignment operator. This list comprises the list of variables to be evaluated. For each variable v in the list L, we call a function to evaluate the expression which defines the variable v. The evaluation function recursively walks over the expression tree and encounters variables and numbers. The evaluation of a number is simply the number itself; the evaluation of a variable is an evaluation of its definition in the symbol table. When a value is obtained as a result of evaluating a definition, the definition is updated to be a value (which will cause the next evaluation of a variable to be trivial, since it will be associated with a number instead of an expression).

6.3 Implementation

6.3.1 The APT Specification File

The APT specification file is encoded directly from the EBNF grammar after it is translated by hand into LL(1) form. We have also defined the end-of-phrase markers and the interface to the semantic routines (whose details will be unveiled later).

```
%PRODUCTIONS
Program -> _Prologue EquationList _Evaluate _Epilogue
EquationList -> Equation ; EquationList
EquationList ->
Equation -> Var := Expression _Equation
Expression ->Term
Expression1 -> AddOp Term _ExprBinOpNew
Expression1 ->
Term -> Factor Term1
Term1 -> MulOp Factor _ExprBinOpNew
Term1 ->
Factor -> Base
Factor1 ->ExpOp Base _ExprBinOpNew
Factor1 ->
Base -> number _ExprLeafNumNew
Base -> Var _ExprLeafVarNew
```

```
Base -> ( Expression ) _Parens
MulOp ->*
MulOp -> /
AddOp -> +
AddOp -> -
ExpOp -> **
Var -> identifier
%ALIASES
:= assign
* mulop
/ divop
+ addop
- subop
** expop
( lparen
) rparen
; semico
%ACTIONS
_PrologueSemPrologue():(Void,0)
_Evaluate SemEvaluate():(Void,0)
_EquationSemEquation(3:Token,1:Expr):(Void,3)
_ExprBinOpNewExprBinOpNew(3:Expr,2:Token,1:Expr):(Expr,3)
_ExprLeafNumNewExprLeafNumNew(1:Token):(Expr,1)
_ExprLeafVarNewExprLeafVarNew(1:Token):(Expr,1)
_ParensSemParens(2:Expr):(Expr,3)
_EpilogueSemEpilogue():(Void,0)
%FIDUCIALS
;
```

6.3.2 The AST Specification File

The AST specification file is directly encoded as a DFSM which accepts the lexical elements described previously. Every lexical action which occurs on a transition from the start state (prominently labelled Start) employs an initial form of the built-in lexical actions to record the starting line and column positions of the token. Also, notice the existence of a rule for the definition of a special token EoF. This token is presently required by the Apt Compiler Toolkit and is assumed to exist by the Apt Parsing Tool. Always include the rule in the AST specification file.

```
Start ":" AssignOp InitAdvance
Start "+" Start InitMatch addop
Start "-" Start InitMatch subop
Start "*" ExpOp InitAdvance
Start "/" Start InitMatch divop
Start alpha Identifier InitAdvance
Start numeric Number InitAdvance
Start "(" Start InitMatch lparen
Start ")" Start InitMatch rparen
Start ";" Start InitMatch semico
Start EndOfFile Start InitIgnore EoF
AssignOp "=" AssignOpMatch assign
AssignOp lambda PushBack Start
```

```
ExpOp "*" ExpOp Match expop
ExpOp lambda ExpOp PushBack mulop
Identifier alphanumeric Identifier Advance
Identifier lambda Identifier PushBack identifier
Number numeric Number Advance
Number "." NumberDot Advance
Number lambda Number PushBack number
NumberDot number NumberDot Advance
NumberDot lambda NumberDot PushBack number
```

6.3.3 The ANT Specification File

The ANT specification file will generate code to build the various data structures which comprise the intermediate representation. The three slots (BinOp, LeafNum, and LeafVar) are all classified as Expr. When this file is processed by the Apt Node Tool, three semantic routines will be generated: ExprBinOpNew, ExprLeafNumNew, and ExprLeafVarNew.

```
include "asttype.h"

node Expr {
    slot BinOp {
        Token operator;
        Expr left, right;
    };
    slot LeafNum {
        Token number;
    };
    slot LeafVar {
        Token name;
    };
}
```

6.3.4 The Remaining Semantic Actions

As the Apt Compiler Toolkit does not yet include a code generation tool (or a tool for the automatic specification of semantics), one must write some code by hand. The code one must write, however, is generally more interesting than the code one must write to do lexical analysis, parsing, and data structures. The five routines which must be written for the equation language include the prologue and the epilogue, the evaluator, the routine which does nothing, and the installation of an equation into the symbol table and the list of variables to be evaluated.

```
#include "eqnodes.h"
#include "eqparse.h"
#include "eqscanner.h"

Queue varQueue;
HashTable varTable;

/* allocate global data structures */
Void SemPrologue(void)
```

```
{
    varQueue = QueueNew();
    varTable = HashTableAVLNew(100, StringHash1, CompareStrings, NULL);
}

/* install definition of var into symbol table */
Void SemEquation(Token var, Expr expression)
{
    if (HashTableAVLLookUpFirst(varTable, var->text) == NULL)
        HashTableAVLInsert(varTable, var->text, expression);
    else
        error("redefined variable", var->text);
}

/* an action which merely returns the expression enclosed in parentheses */
Void SemParens(Expr parenExp)
{
    return parenExp;
}

/* evaluate all variables in the global queue */
Void SemEvaluate(void)
{
    /* see distribution diskettes/tape for the code */
}

/* clean up the global data structures */
Void SemEpilogue(void)
{
    QueueDispose(varQueue, NULL);
    HashTableAVLPtrDispose(varTable);
}
```

6.4 Conclusion

We have successfully designed and implemented our first translator with the Apt Compiler Toolkit. All there is to it involves the parser specification file, the scanner specification file, the node specification file, and the user specification file of actions. All of these files are included in the ACT distribution archive (which is described in the appendix). Supplied with these files is a relatively generic makefile which can be used to automate the compilation and linkage of the translator. All one must do is name the specification and code files appropriately. As all of these are implementation details (which are somewhat specific to operating systems and compiler implementations), we elide all details of the build process here and defer it to the appendix.

We now turn you loose on the Apt Compiler Toolkit and hope you will find it useful for either a course on compilers or your own personal translator. The appendices discuss how to obtain the current version of ACT, how to use ACT under Unix and DOS, how to rebuild ACT with another compiler, and how the generic makefile can be used and tailored to suit your needs.