The Purdue Compiler Construction Tool Set
Version 1.10 Release Notes

ANTLR and **DLG**

August 31, 1993

Terence Parr
Army High Performance Computing Research Center,
University of MN
(parrt@acm.org)

Will Cohen and Hank Dietz
School of Electrical Engineering
Purdue University
(cohenw@ecn.purdue.edu)
(hankd@ecn.purdue.edu)

1. Introduction

This document describes the changes/enhancements in PCCTS since version 1.06. As with the 1.06 release notes, these notes do not constitute the complete reference manual. Unfortunately, the reference manual is in the same condition as it was for the 1.00 release in the Spring of 1992. We are working on a total rewrite of the manual, which might end up in a book consisting of the theory behind practical *k*-token lookahead for $k>1$ (Terence Parr's Ph.D. thesis), ANTLR implementation notes, and the PCCTS user's manual.

The 1.10 release of PCCTS has four main enhancements: fully implemented semantic predicates (`<<...>>?`), infinite lookahead (plus selective backtracking that uses it), increased ANTLR (see `-ck` and `ZZUSE_MACROS` sections) and DLG speed, and the ability to link multiple ANTLR parsers together. A number of bug fixes have been incorporated as well. The tutorials have not been updated much for this release.

To better support our user's, we have established a mailing list called `pccts-users` that you can subscribe to by sending email to `pccts-users-request@arc.umn.edu` with a body of ''`subscribe pccts-users` *your-name*''. Users that have registered with the PCCTS mail server at `pccts@ecn.purdue.edu` have **not** been automatically subscribed. Once you have subscribed, posting a message to the PCCTS community is as simple as sending email to `pccts-users@ahpcrc.umn.edu` (with any Subject: and body). You can also send a body of `help` to `pccts-users-request@ahpcrc.umn.edu` to get help on using the mailing list server.

We have finally agreed on a numbering scheme for PCCTS releases: *x.yz* where *x* reflects the major release number (new tool additions), *y* reflects major new feature additions, and *z* reflects bug fixes and minor feature additions (minor releases).

Partial support for this work has come from the Army Research Office contract number DAAL03-89-C-0038 with the Army High Performance Computing Research Center at the U of MN.

2. Semantic Predicates

The fundamental idea behind semantic predicates has not changed since the 1.06 release — semantic predicates indicate the semantic validity of continuing with the parse or of predicting a particular production. However, we now collect **all** predicates visible to a syntactically ambiguous parsing decision rather than just the first one encountered as in 1.06. In addition, the context of the predicate can be computed and hoisted with the predicates; helpful warning are also generated for incompletely disambiguated parser decisions. The only backward incompatibilities are that parsing does not halt automatically if a semantic validation predicate fails and the `-pr` is obsolete — the specification of a predicate implies that it may be used by ANTLR to validate and disambiguate as it sees fit. In this section we discuss all of these issues.

2.1. Visible Semantic Predicates

Given a syntactically ambiguous parser decision (or, more accurately, a non-deterministic decision), ANTLR attempts to resolve it with semantic information — ANTLR searches for *visible* predicates. A visible predicate is a semantic predicate that could be evaluated without consuming an input token or executing a user action (except initialization actions, which generally define variables). All visible predicates reside on the left edge of productions; predicates not on the left edge can only function as *validation* predicates (see 1.06 release notes). Consider a simple example:

```
a    :    <<pred1>>?  ID α
     |    <<pred2>>?  ID β
     ;
```

Assuming that lookahead information is insufficient to predict productions of rule `a`, ANTLR would incorporate $pred_1$ into the prediction expression for the first production and $pred_2$ into the second prediction expression. Multiple predicates can be hoisted (which may or may not be what you want):

```
decl:   <<pred1>>? var
    |   <<pred2>>? ID
    ;
var :   <<is_var(LATEXT(1))>>? ID
    ;
```

In this case, the prediction expression for production one of rule `decl` would resemble (with context computation turned off — see below):

```
if ( pred1 && is_var(LATEXT(1)) && LA(1)==ID ) {
    var();
}
 ...
```

Here, two visible predicates were found to disambiguate the prediction of the first production of rule `decl` whereas only one was found for the prediction of the second production.

The action of evaluating a predicate in a decision is called *hoisting*. In the first example of this section, predicates were not moved since they reside at the decision point in rule `a`, but technically we say that they were hoisted into the decision. In the second example, $pred_1$ was hoisted from `decl` and `is_var(LATEXT(1))` was hoisted from rule `var` to predict the first production of `decl`.

2.2. Context of Semantic Predicates

In release 1.06, predicates were hoisted without computing and hoisting the context of that predicate. Context is important because, as we saw in the last section, predicates may be evaluated in totally different rules. Imagine a rule that had many alternative productions, two of which were syntactically nondeterministic because of a common lookahead of `ID` (assuming that only one symbol of lookahead is available for simplicity).

```
a   :   A ...
    |   B ...
    |   classname
    |   C ...
    |   varname
    |   D ...
    ;
classname
    :   <<pred1>>? ID
    ;
varname
    :   <<pred2>>? ID
    ;
```

Simply incorporating $pred_i$ into the production prediction expressions for alternatives three and five is not safe for two reasons:

[1] Evaluation of $pred_i$ may cause a program execution error if evaluated on the wrong type of data. $pred_i$ will be evaluated on any input, which is `{A, B, C, D, ID}` in our case — $pred_i$ may ‘‘core’’ if fed non-`ID` token types.

[2] $pred_i$ may give misleading results even if it does not ‘‘core’’. $pred_i$ may return false even though the production is not dependent on the predicate for that token type. For example:

```
a   :   (var | NUM) ...
    |   <<!is_var(LATEXT(1))>>?  ID ...
    ;
var :   <<is_var(LATEXT(1))>>?   ID
    ;
```

The first production will **never** match a `NUM` because `is_var(LATEXT(1))` will always evaluate to false for that token type since numbers are not variables ever (the predicate in `var` is hoisted for use in the decision for rule `a`).

The way to solve both problems is to change $pred_i$ to:

`LA(1)==ID ?` $pred_i$ `: 1`

The `1` merely indicates that if the lookahead is not an `ID` then enable the production for normal parsing — we have no semantic information that establishes the validity or invalidity of the production.

Context computation similar to this is can now be done automatically (`-prc on`). Unfortunately, as mentioned previously in this document, computing full $LL(k)$ lookahead is, in general, an exponential problem; hence, for large grammars you may want to keep this off with `-prc off` (default) and include context tests manually in your predicates. The old `-pr` option to turn on parsing with predicates is now ignored as the specification of a predicate indicates that it should be used.

ANTLR does its best to warn the user when a possibly incompletely disambiguated grammar has been specified. In other words, when a syntactically ambiguous decision is resolved with semantic predicates, all mutually ambiguous productions must have at least one semantic predicate associated with it. For example:

```
a   :   <<pred>>?  ID ...
    |   ID ...
    ;
```

This grammar will yield a warning when run through ANTLR with `-w2` set because semantic information was not provided to indicate the validity of the second production:

```
t.g, line 1: warning: alt 2 of the rule itself has no predicate to resolve ambiguity
```

However, rule `a` will behave correctly because if *pred* fails, the second production will be attempted as the default case (remember that a missing semantic predicate is equivalent to `<<1>>?`). Adding a third production that began with `ID` would not behave correctly as the last `ID`-prefixed production would never be matched.

As a more complicated example, consider the following incorrectly disambiguated grammar:

```
a   :   b
    |   <<pred1>>? ID
    |   <<pred2>>? NUM
    ;

b   :   <<pred3>>? ID
    |   NUM
    ;
```

Rule `a` cannot predict which production to match upon lookahead `ID` or `NUM`. Alternatives 2 and 3 have been disambiguated, but the first production hoists a predicate that only ‘‘covers’’ `ID`’s. As a result, the following message is generated by ANTLR:

```
t.g, line 1: warning: alt 1 of the rule itself has no predicate to resolve ambiguity upon { NUM }
```

This detection is a great help during grammar development.

Ambiguity warnings are now turned off for decisions that have semantic predicates covering all ambiguous lookahead sequences.

2.3. Failure of predicates

Predicates that are not used in disambiguating parsing decisions are called validation predicates. Previously, validation predicates that failed during parsing printed out a message and terminated the parser:

```
if ( !pred ) {fprintf(stderr, "failed predicate: 'pred'\n); exit(1);}
```

The latest release of ANTLR generates a call to a macro that the user may define called `zzfailed_pred()`, which is passed a string representing the predicate that failed:

```
if ( !pred ) {zzfailed_pred("pred");}
```

while this solution is not ideal, it is much better than before.

3. Infinite lookahead and Backtracking

There are a number of grammatical constructs that normal $LL(k)$ recursive-descent parsing cannot handle. The most obvious example would be left-recursion, but left-recursion can be removed by well-known algorithms. The nastiest grammar construct is one in which two alternative productions cannot be distinguished without examining all or most of the production. While left-factoring can handle many of these cases, some cannot be handled due to things like action placement, non-identical left-factors, or alternatives productions that cannot be reorganized into the same rule. The solution to the arbitrarily-large common left-factor problem is simply to use arbitrary lookahead; i.e., as much lookahead as necessary to uniquely determine which production to apply.

ANTLR 1.10 provides two mechanisms for using ‘‘infinite’’ amounts of lookahead. The first is to use semantic predicates in conjunction with a user-defined function that scans arbitrarily ahead using a set of macros provided in this release. The second is a more implicit scheme by which the user can annotate those sections of the grammar, which defy normal LL(k) analysis, with *syntactic predicates*. ANTLR will then generate code that simply tries out the indicated alternative production to see if it would match a portion of the remaining input. If not, the generated parser would try the next viable alternative production. This scheme is a form of selective backtracking (and, hence, can recognize the class of context free languages) where most of a parser is deterministic and only the ‘‘hard’’ parts are done using trial-and-error. As a direct consequence, ANTLR can now generate parsers with the semantic flexibility of LL(k), that are stronger than full LR(k) (in theory), and are nearly linear in complexity; note that the semantic predicates (first introduced in the 1.06 release) can take ANTLR-generated parsers beyond the context-free language limit into the context-sensitive.

We begin this section by introducing the notion of infinite lookahead through an example problem that we solve with semantic and then with syntactic predicates. Following this, we

describe in detail the syntax and use of syntactic predicates, which employ infinite lookahead to perform selective backtracking.

3.1. Examples

This section presents a simple grammar whose productions have common left-factors that we assume, for the sake of demonstration purposes, to be non left-factorable. With nothing but the grammar, ANTLR would be unable to construct a deterministic parser. We first provide a solution by writing a function that explicitly accesses the infinite lookahead buffer to determine which production should be attempted. This solution is efficient, but would become somewhat tedious for the programmer if it had to be done for each such problem in a large grammar. Fortunately, an easier and more concise solution is provided by syntactic predicates, which we also demonstrate using the same grammar.

Consider ML which has multiple assignment and list statements. E.g.,

```
stat:   list Assign list ";"    <<printf("list = list\n");>>
    |   list ";"                <<printf("list\n");>>
    ;
```

This grammar is not $LL(k)$ for any k as `list` can be arbitrarily long. The following grammar using semantic predicates to access the infinite lookahead buffer to explicitly compute which production will be matched.

```
/* example use of the infinite lookahead buffer macros
 * compile with:
 *      antlr list.g
 *      dlg parser.dlg scan.c
 *      cc -Iantlr_includes -DZZINF_LOOK -o list list.c scan.c err.c
 */
#header <<#include "charbuf.h">>

<<
main() { ANTLR(stat(), stdin); }

/* Scan for a "=", but only before a ";" -- return 1 if found, else 0
   This performs the same function as using the syntactic predicate:
        (list Assign list ";")?
   but uses a semantic predicate coupled with the infinite-lookahead feature.
   It is somewhat faster as it does not actually *parse* the "list =", it just
   scans ahead.

   MUST HAVE "ZZINF_LOOK" PREPROCESSOR FLAG DEFINED
   (in #header or on compiler command line)
   */
which()
{
    int i;

    for (i=1; ZZINF_LA_VALID(i); i++)
    {
        if ( ZZINF_LA(i) == Assign ) return 1;
        else if ( ZZINF_LA(i) == Semi ) return 0;
    }
    return 0;
}
>>

#token          "[\ \t]+"       <<zzskip();>>
#token          "\n"            <<zzskip(); zzline++;>>
#token Assign   "="
#token Semi     ";"

stat:  <<which()>>? list Assign list ";" <<printf("list = list\n");>>
    |   list ";"                         <<printf("list\n");>>
    ;

list:   "\(" elem ("," elem)* "\)"
    ;

elem:   ID
    |   INT
    ;

#token ID       "[a-zA-Z]+"
#token INT      "[0-9]+"
```

The infinite lookahead buffer may be accessed with the following macros:

`ZZINF_LA(`*i*`)`

Return the i^{th} token of lookahead relative to the current position. Hence, `ZZINF_LA(1)`..`ZZINF_LA(`*k*`)` are equivalent to `LA(1)`..`LA(`*k*`)`. The difference is that

i can range from the current token of lookahead until the last token of lookahead with `ZZINF_LA(`*i*`)`.

`ZZINF_LATEXT(`*i*`)`

Identical to `ZZINF_LA(`*i*`)` except that the text of the i^{th} token is returned.

`ZZINF_LA_VALID(`*i*`)`

Returns 1 if *i* if at least *i* non-EOF tokens are left in the input stream else it returns 0.

Naturally, the use of infinite lookahead by defining `ZZINF_LOOK` is inconsistent with interactive parsers as the entire input stream is read in before parsing begins.

As mentioned above, this method could be tedious for large grammars, hence, ANTLR provides a more elegant solution. The same problem can be solved with a syntactic predicate by changing rule `stat` in the following way:

```
stat:   ( list Assign list ";" )?       <<printf("list = list\n");>>
    |   list ";"                        <<printf("list\n");>>
    ;
```

Using this implicit method, the need for the semantic predicate and the `which()` function disappears.

Let's now consider a small chunk of the vast C++ declaration syntax. Can you tell exactly what type of object `f` is after having seen the left parenthesis?

```
int f(
```

The answer is ''no.''. Object `f` could be an integer initialized to some previously defined symbol `a`:

```
int f(a);
```

or a function prototype or definition:

```
int f(float a) {...}
```

The following is a greatly simplified grammar for these two declaration types:

```
decl:   type ID "\(" expr_list "\)" ";"
    |   type ID "\(" arg_decl_list "\)" func_def
    ;
```

One notices that left-factoring `type ID "\("` would be trivial because our grammar is so small and the left-prefixes are identical. However, if a user action were required before recognition of the reference to rule `type`, left-factoring would not be possible:

```
decl:   <</* dummy init action so next action is not taken as init */>>
        <<printf("var init\n");>>  type ID "\(" expr_list "\)" ";"
    |   <<printf("func def\n");>>  type ID "\(" arg_decl_list "\)" func_def
    ;
```

The solution to the problem involves looking arbitrarily ahead (`type` could be arbitrarily big, in general) to determine what appears after the left-parenthesis. This problem is easily solved

implicitly by using the new `(...)?` *syntactic predicate*:

```
decl:   (   <<;>> <<printf("var init\n");>> type ID "\(" expr_list "\)" ";"   )?
    |   <<printf("func def\n");>>  type ID "\(" arg_decl_list "\)" func_def
    ;
```

The `(...)?` says that it is impossible to decide, from the left edge of rule `decl` with a finite amount of lookahead, which production to predict. Any grammar construct inside a `(...)?` block is attempted and, if it fails, the next alternative production that could match the input is attempted. This represents selective backtracking and is similar to allowing ANTLR parsers to guess without being ‘‘penalized’’ for being wrong. Note that the first action of any block is considered an init action and, hence, cannot be disabled (by placing it inside `{...}`) since it may define variables; the first action of the block is a dummy action.

At this point, some readers may argue that scanning ahead arbitrarily far, using the infinite lookahead via a semantic or syntactic predicate, renders the parser non-linear in nature. While this is true, the slowdown is negligible as the parser is mostly linear. Further, it is better to have a capability that is slightly inefficient than not to have the capability at all.

3.2. Syntactic Predicates

Just as semantic predicates indicate when a production is valid, *syntactic* predicates also indicate when a production is a candidate for recognition. The difference lies in the type of information used to predict alternative productions. Semantic predicates employ information about the ‘‘meaning’’ of the input (e.g., symbol table information) whereas syntactic predicates employ structural information like normal $LL(k)$ parsing decisions. Syntactic predicates specify a grammatical construct that must be seen on the input stream for a production to be valid. Moreover, this construct may match input streams that are arbitrarily long; normal $LL(k)$ parsers are restricted to using the next k symbols of lookahead. This section describes the form and use of syntactic predicates as well as their implementation.

3.2.1. Syntactic Predicate Form

Syntactic predictions have the form

`( α )? β`

or, the shorthand form

`( α )?`

which is identical to

`( α )? α`

where α and β are arbitrary Extended BNF (EBNF) grammar fragments that do not define new nonterminals. The notation is similar to the `(α)*` and `(α)+` closure blocks already present in PCCTS. The meaning of the long form syntactic predicate is: ‘‘**If α is matched on the input**

stream, attempt to recognize β.'' Note the similarity to the *semantic predicate*:

```
<<α>>? β
```

which means: ''**If α evaluates to true at parser run-time, attempt to match β.**''

Decisions, which are nondeterministic (non-*LL*(*k*) for finite *k*), are resolved via `(..)?` in the following manner:

```
a   :   γ1
    |   γ2
    ...
    |   ( αi )? γi
    ...
    |   γj
    ...
    |   γn
    ;
```

where productions *i* and *j* are mutually nondistinguishable from the left-edge. If production *i* fails, production *j* will be attempted. Typically, the number of syntactic predicates employed is $n-1$ where *n* is the number of mutually nondeterministic productions in a decision; the last production is attempted by default.

When a production to be predicted must be predicted with itself (nothing less sophisticated is sufficient) or when efficiency is not a major concern, the short form is used:

```
a   :   γ1
    |   γ2
    ...
    |   ( γi )?
    ...
    |   γj
    ...
    |   γn
    ;
```

3.2.2. Modified *LL*(*k*) Parsing Scheme

Decisions that are not augmented with syntactic predicates are parsed deterministically with finite lookahead up to depth *k* as is normal for PCCTS-generated parsers. When at least one syntactic predicate is present in a decision, rule recognition proceeds as follows:

[1] Find the first *viable* production; i.e. the first production in the alternative list predicted by the current finite lookahead, according to the associated finite-lookahead prediction-expression.

[2] If the first element in that production is not a syntactic predicate, predict that production and go to [4] else attempt to match its predicting grammar fragment.

[3] If the grammar fragment is matched, predict the associated production and go to [4] else find the next viable production and go to [2].

[4] Proceed with the normal recognition of the production predicted in [2] or [3].

For successful predicates, both the predicting grammar fragment and the remainder of the production are actually matched, hence, the short form, `(α)?`, actually matches α twice — once to predict and once to apply α normally.

3.2.3. Syntactic Predicate Placement

Syntactic predicates may only appear as the first element of a production because that is the only place decisions are required. For example, the `(..)?` block in the first production of following grammar has little utility.

```
a   :   γ1 ( α )? β
    |   γ2
    ...
    |   γn
    ;
```

There is no question that β is to be matched after γ_1 and trying to predict this situation is redundant.

Syntactic predicates may appear on the left edge of any production within any subrule, not just in productions at the rule block level.

3.2.4. Nested Syntactic Predicate Invocation

Because syntactic predicates may reference any defined nonterminal and because of the recursive nature of grammars, it is possible for the parser to return to a point in the grammar which had already requested backtracking. This nested invocation poses no problem from a theoretical point of view, but can cause unexpected parsing delays in practice.

3.2.5. Grammar Fragments within Syntactic Predicates

The grammar fragments within `(α)?` may be any valid PCCTS production right-hand-side; i.e. any expression except new nonterminal definitions. α may contain semantic actions and semantic predicates, although only the semantic predicates will be executed during prediction.

3.2.6. Efficiency

In terms of efficiency, the order of alternative productions in a decision is significant. Productions in a PCCTS grammar are always attempted in the order specified. For example, the parsing strategy outline above indicates that the following rule is most efficient when γ_1 is less complex than γ_2.

```
a   :   (γ1)?
    |   γ2
    ;
```

Any parsing decisions made inside a `(..)?` block are made deterministically unless they themselves are prefixed with syntactic predicates. For example,

```
a   :   ( (A)+ X | (B)+ X )?
    |   (A)* Y
    ;
```

specifies that the parser should attempt to match the nonpredicated subrule

```
    (   (A)+ X
    |   (B)+ X
    )
```

using normal the normal finite-lookahead parsing strategy. If a sentence recognizable by this grammar fragment is found on the input stream, then restore the state of the parser to what it was before the predicate invocation and parse the grammar fragment again; else, if the attempt failed, apply the next production in the outer block:

```
(A)* Y
```

3.2.7. Resolving Ambiguous C++ Statements

Quoting from Ellis and Stroustrup [‘‘The Annotated C++ Reference Manual,’’ Margaret A. Ellis and Bjarne Stroustrup, Addison Wesley Publishing Company; Reading, Massachusetts; 1990],

> ‘‘There is an ambiguity in the grammar involving *expression-statements* and *declarations*... The general cases cannot be resolved without backtracking... In particular, the lookahead needed to disambiguate this case is not limited.’’

The authors use the following examples to make their point:

```
T(*a)->m=7;      // expression-statement
T(*a)(int);      // declaration
```

Clearly, the two types of statements are not distinguishable from the left as an arbitrary amount of symbols may be seen before a decision can be made; here, the `->` symbol is the first clue that the first example is a statement. Quoting Ellis and Stroustrup further,

> ‘‘In a parser with backtracking the disambiguating rule can be stated very simply:
> [1] If it looks like a *declaration*, it is; otherwise
> [2] if it looks like an *expression*, it is; otherwise
> [3] it is a syntax error.’’

The solution in PCCTS using syntactic predicates is simply:

```
stat:   (declaration)?
    |   expression
    ;
```

The semantics of rule `stat` are exactly that of the quoted solution. The production `declaration` will, however, be recognized twice upon a valid declaration and once upon an expression

to decide that it is not a declaration.

3.2.8. Revisiting the ML Example

To illustrate the utility of the full form of syntactic predicates, reconsider the grammar for the ML-style statements provided in the example section above:

```
stat:   list "=" list ";"
    |   list ";"
    ;
```

Rule `stat` is not *LL* because `list` could be arbitrarily long and, hence, predicting which production to apply beforehand is impossible with a finite lookahead depth. There are two solutions in using syntactic predicates, one more efficient than the other. The first method is, as before, to specify:

```
stat:   (list "=" list ";")?
    |   list ";"
    ;
```

However, this specification unnecessarily matches the `list` following the assignment operator twice. A more efficient, but functionally equivalent, specification is as follows:

```
stat:   (list "=")? list "=" list ";"
    |   list ";"
    ;
```

This description indicates that, as soon as the `"="` has been seen, the first production is uniquely predicted.

3.2.9. Syntactic Predicates Effect on Grammar Analysis

ANTLR still constructs normal $LL(k)$ decisions throughout predicated parsers. Only when necessary are arbitrary lookahead predictors used. Constructing $LL(k)$ parsers is an exponential problem that ANTLR goes to great lengths to avoid or reduce in size on average. Unfortunately, for large grammars and k values of more than 2 or 3 ANTLR can take an impractical amount of time. Part of the benefit of `(..)?` blocks is that, by definition, they defy $LL(k)$ analysis. Hence, the exponential, full $LL(k)$ grammar analysis is turned off for any production beginning with a syntactic predicate. In its place, a linear approximation to $LL(k)$ analysis, called $LL^1(k)$, is used. This reduces the number of times that arbitrary lookahead `(..)?` blocks are attempted unnecessarily, though no finite lookahead decision is actually required as the arbitrary lookahead mechanism will accurately predict the production.

If the current finite lookahead can predict which production to apply, syntactic predicates are not evaluated. For example, referring to the C++ declaration versus expression grammar example above, if the current input token were `42`, rule `stat` would immediately attempt the second production — `expression`. On the other hand, if the current input token were `ID`, then the `declaration` rule would be attempted before attempting `expression`. If neither productions successfully match the input, a syntax occurs.

When constructing finite lookahead sets, the grammar fragment within the `(..)?` block is ignored. In other words, $FIRST_k((\alpha)?\ \beta)$ is $FIRST_k(\beta)$.

3.2.10. The Effect of Nondeterminism upon Translation and Semantic Predicates

Syntactic predicates are, by definition, not guaranteed to match the current input. Therefore, actions with side-effects, for which no ‘‘undo’’ exists, cannot be executed during nondeterministic syntactic prediction (‘‘guess’’ mode). This section describes how ANTLR handles the execution of user-supplied actions and semantic predicates.

3.2.10.1. The Effect upon User Actions

PCCTS language specifications do not allow the execution of any semantic action during a syntactic prediction as no undo mechanism exists; this conservative scheme avoids affecting the parser state in an irreversible manner. The only exception to this rule is that initialization actions, which usually define variables visible to the entire rule/function, are not enclosed in `if {..}` statements to ‘‘gate’’ them out; hence, initialization actions with side effects must be avoided by the PCCTS user.

3.2.10.2. The Effect upon Semantic Predicates

Semantic predicates are always evaluated because they are restricted to side-effect-free expressions. During arbitrary lookahead prediction, the semantic predicates that are evaluated must be functions of values computed when actions were turned on. For example, if your grammar has a predicate that examines the symbol table, all symbols needed to direct the parse during prediction must be entered into the table before prediction has begun. Consider the following grammar fragment which recognizes simplified C declarations.

```
decl:   "typedef" type declarator ";"                    /* define new type */
    |   ( type declarator "\{" )? type declarator func_body   /* define function */
    |   type declarators ";"                             /* def/decl var(s) */
    ;

type:   built_in_type
    |   <<is_type(LATEXT(1))>>? ID
    ;

declarator
    :   ...
        /* recognizes a declarator such as ‘‘array[3]’’ */
        /* add symbols, both types and vars, to the symbol table */
    ;
```

This rule is unnecessarily inefficient, but will illustrate the evaluation of semantic predicates during nondeterministic prediction. For the purposes of our discussion, we restrict new types to be introduced using a `typedef` (structures and unions are not allowed). Consider the recognition of the two sentences:

```
typedef int My_int;
My_int i;
```

The first production of rule `decl` will match the first sentence, adding `My_int` to the symbol table as a type name. Production two of `decl` attempts to match the second sentence with its syntactic predicate. Rule `type` is entered, which evaluates `is_type(LATEXT(1))` (where `is_type()` is some user-defined function that looks up its symbol argument in the symbol table and returns true if that symbol is defined and is a type). Because the text of the current token of lookahead, `My_int`, is a valid type, the predicate evaluates to true. Production two of `type` is applicable semantically and is, therefore, applied. After consuming `My_int`, the parser successfully applies `declarator` to `i`. The next input token is `;` which does not match . The nondeterministic prediction fails and production three is predicted by default and is applied.

The second production of rule `decl` could not be rewritten as

```
( type declarator func_body  )?  /* define function */
```

because, presumably, a func_body could define new types. The actions that add these new types to the symbol table would not be executed, however, as the parser would be in nondeterministic mode. Although the semantic predicates would be evaluated correctly, the symbol table would not hold the information necessary to parse the function body during nondeterministic prediction. Also, this revision is very inefficient as it would match the entire function, which could be large, twice.

3.2.11. Comparing the Use of Semantic and Syntactic Predicates

Language constructs exists that are totally ambiguous syntactically, but easily distinguishable semantically. For example, array references and function calls in Fortran are identical syntactically, but very different semantically. The associated grammatical description is non-*LL*(*k*), non-*LALR*(*k*), and non-context-free; not even backtracking or infinite lookahead will help this problem.

```
expratom:   ID "\(" expr_list "\)"
       |   ID "\(" expr_list "\)"
       ...
       ;
```

where `expr_list` is some rule matching a comma-separated expression list. Putting `(..)?` around the first alternative production will not change the fact that both productions match the same sentence. However, semantic predicates may be used to semantically disambiguate the rule:

```
expratom:   <<isVar(LATEXT(1))>>?    ID "\(" expr_list "\)"
       |   <<isFunc(LATEXT(1))>>?   ID "\(" expr_list "\)"
       ...
       ;
```

3.2.12. Implementation

The discussion thus far has described the functionality of syntactic predicates, but their implementation is an equally important topic so that users can understand the new ANTLR parsing mechanism (e.g., so that users can follow along in a debugger while tracking down bugs in their grammar).

Because productions are assumed to be attempted in the order specified, a nested if-then-else structure is generated. To illustrate the integration of syntactic predicates into the normal ANTLR code generation scheme, consider the following abstract grammar.

```
a    :    γ1
   ...
     |    ( αi )? γi
   ...
     |    ( αj )? γj
   ...
     |    γm
     ;
```

ANTLR generates the following, ‘‘slightly sanitized’’, code:

```
a()
{
    zzGUESS_BLOCK
    if ( (τ1, ..., τk) ∈ LOOKk(γ1) ) {
        match γ1;
    }
    else {
        zzGUESS
        if ( !zzrv && (τ1, ..., τk) ∈ LOOKk(γi) ) {
            match αi;
            zzGUESS_DONE
            match γi;
        }
        else {
            if ( zzguessing ) zzGUESS_DONE;
            zzGUESS
            if ( !zzrv && (τ1, ..., τk) ∈ LOOKk(γj) ) {
                match αj;
                zzGUESS_DONE
                match γj;
            }
            else {
                if ( zzguessing ) zzGUESS_DONE;
                if ( (τ1, ..., τk) ∈ LOOKk(γm) ) {
                    match γm;
                }
                else goto fail;
            }
        }
    }
    return;
fail:
    if ( zzguessing ) {zzGUESS_FAIL;}
    gen syntax error message;
    resynch parser;
}
```

where $LOOK_k(\gamma)$ is the set of lookahead k-tuples that predict γ. This notation is used as a convenience here whereas ANTLR generates decisions that use as little lookahead as possible in practice.

The macros/variables themselves are defined as follows:

`zzGUESS_BLOCK`

Define a block of memory to hold the current parser state and the return value of `setjmp()`, `zzrv`.

`zzGUESS`

Save the current parser state, turn on guessing mode and call `setjmp()` to record the current C run-time stack state. The result of `setjmp()` is placed into `zzrv`.

`zzGUESS_FAIL`

Long jump — restore the C run-time stack to the state it held before guessing began.

`zzGUESS_DONE`

Restore the parser state to the previously saved contents.

`zzguessing`

This variable is 1 if a prediction is currently underway and 0 when normal parsing is proceeding. User actions are turned off when this variable is 1.

`zzrv`

This variable is the result of doing the `setjmp()` call, which returns 0 always. When a `longjmp()` occurs, the C run-time stack will be reset to the state held at the time of the `setjmp()` and `zzrv` will be set to a nonzero value. In the view of the C program, it will appear as if the `setjmp()` has returned without ever having attempted the code in the `if` following it; execution continues past the `if` the second time.

All semantic actions except initialization actions are enclosed in

```
zzNON_GUESS_MODE {
     user-defined-semantic-action;
}
```

so that they can be ‘‘turned off’’ during a prediction. `zzNON_GUESS_MODE` is defined as follows:

```
if ( !zzguessing )
```

The effect of this type of code generation is that a stack of parser states is maintained such that nested nondeterministic predictions can be made.

As an optimization, when the prediction grammar fragment for a production is regular, simpler recognition schemes could be used.

4. DLG Enhancements

There have been a number of changes to dlg from 1.06 to 1.10. The main difference is that DLG execution speed is up to 7 times faster than the 1.06 version. A `-Wambiguity` option has been added to indicate where ambiguities in DLG specifications exists. It numbers the expressions and prints out for an accept state the possible expressions that could be recognized. Also, a macro called `ANTLRs()` has been added that behaves just like `ANTLR()` except that it accepts input from a string rather than a stream:

```
#define ANTLRs(rule, string) {...}
```

5. Linear-Approximation Lookahead

ANTLR-generated parsers predict which rule alternative to match by examining up to *k* symbols of lookahead. Unfortunately, computing (during ANTLR grammar analysis) and examining (during parser execution) the set of possible *k*-sequences is an exponentially large problem. A linear approximation to this full lookahead exists that requires linear time to compute and to test; further, this approximation handles the majority of parsing lookahead decisions. To avoid the, possibly exponential, computation of full lookahead, ANTLR attempts to use the linear approximation first — computing full lookahead as a last resort. The reason that ANTLR

occasionally goes ‘‘off the deep end’’ when analyzing some big grammars is that ANTLR found a parsing decision that could not be solved with the approximate lookahead and required exponential time to compute the full lookahead.

Because the approximation has linear time and space complexity, its lookahead depth can be much deeper than that of the full lookahead. Consequently, the approximate lookahead is sometimes stronger than the full lookahead because it can look farther ahead without consuming an impractical amount of system resources. We have added an ANTLR option, called `-ck` *n*, that allows the user to specify how deep the linear approximation analysis should go before giving up and trying the full lookahead computation. This new feature is best described with an example:

```
a   :   (A B|C D) E
    |   A D F
    ;
```

The full $LL(2)$ lookahead (as would be computed by ‘‘`antlr -k 2 ...`’’) is summarized in the following table

$LL(2)$	
Lookahead	Alternative
A B	1
C D	1
A D	2

whereas the linear approximate lookahead, denoted $LL^1(2)$ (as would be computed by ‘‘`antlr -ck 2 ...`’’), is

$LL^1(2)$	
Lookahead	Alternative
{A,C} {B,D}	1
{A} {D}	2

where lookahead `{A,C} {B,D}` implies that the first symbol of lookahead can be either `A` or `C` and the second can be either `B` or `D`; this lookahead therefore matches the set of sequences `{A B, A D, C B, C D}`, which is like the cross product of the sets (note the loss of sequence information, which results in the approximation). The decision is $LL(2)$, but is not $LL^1(2)$ because the sequence `A D` predicts both alternatives (i.e., `A` can be seen first by both and `D` can be seen second). However, if ANTLR were allowed to look 3 symbols ahead — $LL^1(3)$ — the linear approximation would be sufficient and the complex $LL(3)$ would not be computed. The $LL^1(3)$ information (‘‘`antlr -ck 3`’’) is summarized in the following table:

$LL^1(3)$	
Lookahead	Alternative
{A,C} {B,D} {E}	1
{A} {D} {F}	2

Notice that, now, the third symbol of lookahead alone can uniquely identify which alternative to predict.

Let's augment our example to have one $LL(2)$ decision and one $LL^1(3)$ decision:

```
a   :   (A B | C D) E   /* LL^1(3) */
    |   A D F b
    ;
b   :   (A B | C D) Z   /* LL(2) */
    |   A D Z
    ;
```

Although $LL(3)$ (''`antlr -k 3 ...`'') handles both the $LL^1(3)$ and $LL(2)$ decisions, we can make ANTLR and the resulting parser more efficient by specifying ''`antlr -ck 3 -k 2 ...`''. The resulting parser decisions are illustrated in the following (sanitized) code fragment:

```
a()
{
    /* there are no sequence comparisons for rule 'a' because LL^1(3)
     * is sufficient and full LL(3) analysis is not invoked
     */
    if ( LA(1)∈{A,C} && LA(2)∈{B,D} && LA(3)==E ) {
        if ( (LA(1)==A) ) {
            zzmatch(A); zzCONSUME;
            zzmatch(B); zzCONSUME;
        }
        else if ( (LA(1)==C) ) {
            zzmatch(C); zzCONSUME;
            zzmatch(D); zzCONSUME;
        }
        zzmatch(E); zzCONSUME;
    }
    else if ( (LA(1)==A) && (LA(2)==D) && (LA(3)==F) ) {
        zzmatch(A); zzCONSUME;
        zzmatch(D); zzCONSUME;
        zzmatch(F); zzCONSUME;
        b();
    }
}
```

```
b()
{
    /* LL(2) decision */
    if ( (LA(1)==A&&LA(2)==B) || (LA(1)==C&&LA(2)==D) ) {
        if ( (LA(1)==A) ) {
            zzmatch(A); zzCONSUME;
            zzmatch(B); zzCONSUME;
        }
        else if ( (LA(1)==C) ) {
            zzmatch(C); zzCONSUME;
            zzmatch(D); zzCONSUME;
        }
    }
    else if ( LA(1)==A&&LA(2)==D ) {
        zzmatch(A); zzCONSUME;
        zzmatch(D); zzCONSUME;
        zzmatch(Z); zzCONSUME;
    }
}
```

These examples are small and, hence, the savings are not apparent, but the ‘‘compressed’’ approximation to full lookahead can be used to reduce the ANTLR execution time and resulting parser speed/size.

6. Faster Compilation of ANTLR-Generated Parsers

Previous versions of ANTLR used macros rather than function calls for many operations during parsing. Because the macros were invoked numerous times, compilation of these files was slow and generated large object files. The operations are now, by default, function calls which makes compilation about 2 times as fast and results in object files about half as large. The macros can be used if necessary by defining `ZZUSE_MACROS` on the compile line (`-DZZUSE_MACROS`).

7. Linking Together Multiple ANTLR Parsers

Because of the lack of sophisticated ‘‘information hiding’’ in C, many ANTLR program symbols are globally visible and, hence, linking multiple ANTLR-generated parsers together would cause many name collisions. To overcome this, we have introduced a new ANTLR directive:

```
#parser "my_parser_name"
```

which prefixes all global, externally visible symbols with prefix `my_parser_name_` (remember this when debugging ANTLR parsers). This, clearly, renders the previous ‘‘generate prefix’’ option (`-gp`) obsolete. Variables, functions and rule names are remapped through the inclusion of a file called `remap.h`, which is automatically generated by ANTLR when it encounters a `#parser` directive. In the future, we expect this to be the name of a C++ object of some class `Parser`; variables and functions will be referenced as `my_parser_name.var_or_func`.

Consider the following ANTLR example. Files `t.g` and `t2.g` are identical except for the parser name.

File `t.g`

```
#header <<#include "charbuf.h">>

#parser "t"

<<
void parse_t()
{
    ANTLR(a(), stdin);
}
>>

#token "[\ \t\n]"       <<zzskip();>>

a : INT INT
  ;

#token INT "[0-9]+"
```

File `t2.g`

```
#header <<#include "charbuf.h">>

#parser "t2"

<<
void parse_t2()
{
    ANTLR(a(), stdin);
}
>>

#token "[\ \t\n]"       <<zzskip();>>

a : INT INT
  ;

#token INT "[0-9]+"
```

File `main.c`

```
#include <stdio.h>

extern void parse_ter();
extern void parse_ter2();

main()
{
    parse_ter();
    parse_ter2();
}
```

File `makefile`

```
DLG_FILE = parser.dlg
ERR_FILE = err.c
HDR_FILE = stdpccts.h
TOK_FILE = tokens.h
K = 1
ANTLR_H = ../h
BIN = ../bin
ANTLR = ../bin/antlr
DLG = $(BIN)/dlg
CFLAGS = -I. -I$(ANTLR_H) -g
AFLAGS = -fe err.c -fl parser.dlg -ft tokens.h -fr remap.h -fm mode.h  \
         -gt -gk
AFLAGS2= -fe t2_err.c -fl t2_parser.dlg -ft t2_tokens.h -fr t2_remap.h \
         -fm t2_mode.h -gt -gk
DFLAGS = -C2 -i
GRM = t.g
SRC1 = scan.c t.c err.c
SRC2 = t2.c t2_scan.c t2_err.c main.c
OBJ1 = scan.o t.o err.o
OBJ2 = t2.o t2_scan.o t2_err.o main.o
CC=g++

t: $(OBJ1) $(OBJ2)
    $(CC) -o t $(CFLAGS) $(OBJ1) $(OBJ2)

t.o : mode.h tokens.h t.g

scan.c mode.h : parser.dlg
    $(DLG) $(DFLAGS) parser.dlg scan.c

t.c parser.dlg tokens.h : t.g
    $(ANTLR) $(AFLAGS) t.g

t2.o : t2_mode.h t2_tokens.h t2.g

t2_scan.c t2_mode.h : t2_parser.dlg
    $(DLG) $(DFLAGS) -m t2_mode.h t2_parser.dlg t2_scan.c

t2.c t2_parser.dlg t2_tokens.h : t2.g
    $(ANTLR) $(AFLAGS2) t2.g
```

The input to the parser is 4 integers because each of the invoked parsers matches 2.

The preprocessor symbol `zzparser` is set the parser name string specified in the `#parser "name"` directive.

WARNING: the remapping of symbols to avoid collisions is not a foolproof system. For example, if you have a rule named `type` and a field in a structure named `type`, the field name will get renamed as well — this is the price you pay for being able to link things together without C++.

8. Creating Customized Syntax Error Routines

Many users have asked how to create their own `zzsyn()` error handling routine. Here's how:

[1] Make new `zzsyn()` with same parameters.

[2] Define the preprocessor symbol `USER_ZZSYN` on the compile line (`-DUSER_ZZSYN`).

9. Lexical Changes to ANTLR Input

The manner in which ANTLR interprets user actions has changed. Strings, character literals, and C/C++ comments are now totally ignored. For example,

```
<<
    // nothing in here is examined $1 ' "
    /* or in here ' " $ #[jfd] '"'''" */
    '"' // that's a character
    "'" // that's an apostrophe
    "$1 is", $1 /* $1 inside string is ignored */
>>
```

As a result of this change, you may experience a slight difference in how ANTLR treats your actions. Comments inside actions are still passed through to the parser.

C++ comments are now accepted outside of actions as well:

```
// this rule does nothing
a : ;
```

Watch out for this:

```
  ...
  << // a comment >>
  ...
a : A
  ;
```

The C++ style comment in side the action will scarf til end of line and ignore the `>>` end action symbol. This could be avoided, but I'm feeling lazy just now.

10. New ANTLR Options

Release 1.10 introduces the following ANTLR command-line options:

`-` ANTLR now accepts input from `stdin` by using the `-` option; e.g.,

```
antlr -
```

A file called `stdin.c` is created as the output parser.

-ck *n* Use up to *n* symbols of lookahead when using compressed (linear approximation) lookahead. This type of lookahead is very cheap to compute and is attempted before full LL(k) lookahead, which is of exponential complexity in the worst case. In general, the compressed lookahead can be much deeper (e.g, `-ck 10`) than the full lookahead (which usually must be less than 4).

`-fm` *mode_file*
Rename file with lexical mode definitions, *mode.h*, to file.

-fr *file*Rename file which remaps globally visible symbols, *remap.h*, to file. This file is only created if a `#parser` directive is found.

-prc on

Turn on the computation and hoisting of predicate context.

-prc off

Turn off the computation and hoisting of predicate context. This option makes 1.10 behave like the 1.06 release with option **-pr** on (default).

-w1 Set low warning level. Do not warn if semantic predicates and/or (...)? blocks are assumed to cover ambiguous alternatives.

-w2 Ambiguous parsing decisions yield warnings even if semantic predicates or (...)? blocks are used. Warn if `-prc on` and some lookahead sequences are not disambiguated with a hoisted predicate.

11. ANTLR Generates ‘‘Androgynous’’ Code

The distribution source of 1.06 PCCTS was generated using 1.06 on a 32-bit machine. Unfortunately, the source code dumped bit sets to arrays of `unsigned`’s according to the word size of the machine that generated the parser — regardless of the word size of the various target machines. To overcome this, ANTLR always dumps its bit sets as arrays of `unsigned char`, which are 8 bits (or more) on any machine that we’d ever want to work on. As a result, ANTLR itself should bootstrap on any machine with a C compiler a enough memory. We have gotten it to compile with 16-bit Microsoft and Borland C on the PC with only a few whimpers. The makefiles in the ANTLR and DLG directories have sections for each of the various compilers.

12. Printing out grammars

Using the `-p` option generates grammar listings that are somewhat nicer.

13. C Grammar Changes

The C grammar example has been augmented with a `-both` option that prints out both K&R and ANSI C prototypes for functions defined in the input file. E.g.

```
% proto -both
void f(a,b)
int a;
char *b;
{;}
^D
 void
#ifdef __STDC__
f( int a, char *b )
#else
f( a, b )
 int a;
 char *b;
#endif
```

Functions that already employ ANSI C style argument definitions are handled as well.

14. C++ Now Compiles ANTLR Itself

We have modified the source code of ANTLR to compile under C++. It is not written to take advantage of C++'s extensions to C, however, except in rare instances. C++'s stricter type checking motivated the modification.

15. New Preprocessor Symbol

ANTLR now generates a `#define` called `ANTLR_VERSION` that is set to the version of ANTLR that generated the parser. For this release, you will see:

```
#define ANTLR_VERSION 110
```

in the output files, which is an integer equivalent of the version number.

16. Attribute Warning

A number of users have had trouble with the `charptr.h` attributes. Please note that they do **not** make copies and that the memory is freed after the scope exits. For example, this is **wrong** because the memory for the `$1` attribute of `A` or `B` in the `(...)` scope will be freed upon exit even though `$0` will still point to it.

```
#header<<#include "charptr.h">>

<<
#include "charptr.c"
main() { ANTLR(a(),stdin); }
>>

#token "[\ \t\n]"       << zzskip(); >>

a: "ick" ("A" << $0=$1; >>| "B" << $0=$1; >>) "ugh"
   << printf("$1, $2, $3 are %s, %s, %s\n",$1, $2, $3); >>
 ;
```

One should make a copy of the local attribute (or use `charbuf.h`) as the mem is freed at the end of the scope (`$0=strdup($1);`).

17. Generation of Line Information

The normal form of line information is:

 # line_number "file"

However, many compilers, such as Borland C, prefer it as

`#line` *line_number "file"*

This can be easily changed by looking file `generic.h` in the `antlr` directory for the following:

```
/* User may redefine how line information looks */
#define LineInfoFormatStr "# %d \"%s\"\n"
```

Simply change it as your compiler wants it and recompile the antlr source.

18. Incompatibilities

There should be very few incompatibilities with your 1.06-based grammars. Should you find any please let us know.

1.06 semantic predicates were not hoisted into parsing decisions without the `-pr` flag (now obsolete). In 1.10, the use of a predicate indicates that it may be hoisted.

Semantic predicates used to halt parser upon failure whereas 1.10 does not.

The interpretation of strings, character literals, and comments are now handled differently; see above.

19. Future Directions

This section briefly describes some of the future enhancements either being discussed, planned, or developed.

- A graphical user interface is planned for ANTLR grammars that will allow the simultaneous display/manipulation of BNF and syntax diagram representations of user grammars.
- A source-to-source translator-generator called SORCERER is in prototype form. It's input looks like ANTLR and is integrated so that one description will contain lexical, syntactic, and tree-translation information.
- A number of groups are working on a C++ grammar. Things are starting to heat up as it is pretty much certain that 1.10 ANTLR is the minimum necessary system to parse C++.
- A code-generator generator, called PIGG, is in prototype form.
- An assembler generator is in prototype form.

- DLG backtracking will be added.
- A new ‘‘magic’’ token type, ‘‘.’’, will be introduced which means ‘‘match any single token.’’
- A new operator will be introduced, ‘‘~’’, which will allow constructs like `~(A|B|C)` — implying ‘‘match a single token **not** from the set {A, B, C}.’’
- A new ANTLR directive will be introduced:

  `#tokclass` *name* `{` *token_list* `}`

  that creates a set of tokens like the `#errclass` directive, but one which can be referenced in the grammar. For example:

```
#tokclass AOP { "–" "+" }
#tokclass MOP { "/" " }
#tokclass OP  { AOP MOP }
 ...
e : e1 ( AOP e1 )* ;
e1: e2 ( MOP e2 )* ;
 ...
```

- Simple left-factoring will be introduced to remove identical left factors from alternative productions (assuming user actions do not interfere).
- A version of ANTLR (called ANTLR-lite?) is being considered that would accept most ANTLR description syntax, delay grammar analysis to run time (where it could be done much more quickly — with nonexponential complexity), scan for tokens with an NFA regular-expression interpreter rather a DFA, and place all output in one nice little file. The reduction in parser size would be substantial, but at a parser run-time cost.

20. Portability

PCCTS 1.10 is known to compile ‘‘out of the box’’ on the following machines and/or operating systems:

[1] DECSTATION 5000

[2] SGI, Running IRIX 4.0.5

[3] Sun SparcStation (cc, gcc, g++, Cfront)

[4] DOS and OS/2, Microsft C 6.0, 16 bit

[5] DOS and OS/2, Borland C/C++, 16 bit

[6] OS/2, IBM C-Set/2, 32 bit

[7] VAX C under VMS

[8] Linux 0.99, gcc/g++

[9] NeXT box

[10] Amiga, AmigaDOS--SAS/C Development System V 5.10b

21. Beta Testers

The following is a group of persons (listed alphabetically) that, in some way, have helped shape and/or debug the latest release of PCCTS.

[1] Steven Anderson, (sea@ahpcrc.umn.edu)

[2] Douglas B. Cuthbertson, (cuthbertsond@gw1.hanscom.af.mil)

[3] Peter Dahl, (dahl@mckinley.ee.umn.edu)

[4] Ed Harfmann, (mdbs!ed@dynamo.ecn.purdue.edu)

[5] Randy Helzerman, (helz@ecn.purdue.edu)

[6] Stephen Hite, (shite@sinkhole.unf.edu)

[7] Dana "Muck" Hoggatt, (mdbs!muck@dynamo.ecn.purdue.edu)

[8] Roy Levow, (roy@gemini.cse.fau.edu)

[9] John Mejia, (mejia@mckinley.ee.umn.edu)

[10] David Poole, (dpoole@nitrogen.oscs.montana.edu)

[11] Russell Quong, (quong@ecn.purdue.edu)

[12] Aaron Sawdey, (sawdey@mckinley.ee.umn.edu)

[13] Fred Scholldorf, (scholldorf@nuclear.physics.sunysb.edu)

[14] Sumana Srinivasan, (Sumana_Srinivasan@next.com)

[15] Ariel Tamches, (tamches@cs.wisc.edu)