Introductory Tutorial

PCCTS 1.0x

Terence Parr, Hank Dietz, Will Cohen

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
Fall 1992

`parrt@ecn.purdue.edu`
`hankd@ecn.purdue.edu`
`cohenw@ecn.purdue.edu`

The Purdue Compiler-Construction Tool Set (PCCTS) is a set of public domain software tools designed to facilitate the implementation of compilers and other translation systems. In many ways, PCCTS is similar to a highly integrated version of YACC and LEX; where ANTLR (ANother Tool for Language Recognition) corresponds to YACC and DLG (DFA-based Lexical analyzer Generator) functions like LEX. However, PCCTS has many additional features which make it easier to use for a wide range of translation problems.

This document introduces the basic functionality of PCCTS by example. The user need not be familiar with parsing theory or other compiler tools, but any familiarity reduces the learning curve substantially. The PCCTS reference manual is a necessary supplement to this tutorial as information here regarding PCCTS structures and operation is incomplete.

1. Introduction

PCCTS allows the user to describe languages (e.g. programming language, OS shell, game, editor); from such a description, a C program is generated that recognizes and, optionally, translates phrases in that language. The user must specify the following:

(i) How the input stream is to be broken up into lexemes (tokens) which comprise the vocabulary of the language.

(ii) How the tokens are to be grouped; i.e. what structure/grammar is to be applied to the token stream.

(iii) C actions which perform a user-specified translation. Along with this specification, the user must also describe token attributes—objects that actions use to communicate with the lexical analysis phase of translation.

Similarly, this tutorial is broken up into sections on lexical analysis, syntactic analysis, and actions/translation.

2. Lexical Analysis

Before understanding a phrase in English, one must separate the stream of characters into a stream of words; e.g. the phrase: ‘‘thisisveryhardtoread’’ accentuates this fact—recognition cannot easily be done from a character stream, only from word/token streams.

Compilers and other translators are very strict about this ‘‘tokenization’’ and generally describe tokens via *regular expressions*—expressions that describe sets of character sequences. The regular expressions are, in fact, language descriptions as well. For example, `hello` is a regular expression that recognizes a sequence of five characters; namely, the word: ‘‘hello’’. To inform PCCTS that ‘‘hello’’ is to be a word in the vocabulary of your language, the following description would be placed in your grammar file.

```
#token LABEL "hello"
```

where `LABEL` is some label (C `#define`) that you want associated with that token. To test regular expressions in PCCTS, let us form a simple, complete description which recognizes ‘‘hello’’ (we will use this description as a base for all examples in this section):

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

<<main() { ANTLR(a(), stdin); }>>

#token WORD "hello"

a : WORD ;
```

This is a minimal description in that it contains everything needed for PCCTS to generate an executable (actually, to generate all C files needed for the C compiler to generate an executable). The `#header <<...>>` instruction informs PCCTS that the C code inside the `<<...>>` action is necessary to define attributes and to compile the actions found elsewhere; for this section, we will ignore its significance. The second action gives a main program that specifies where C is to begin execution. It contains one statement which asks ANTLR to begin parsing at rule `a`. The third instruction defines a token `hello`. The fourth component of this description is a rule definition. Rules definitions have the form:

rule`:` $\textit{alternative}_1$ `|` $\textit{alternative}_2$ `| ... |` $\textit{alternative}_n$ `;`

where each alternative is a sequence of grammatical structures that are to be matched— one of possible structures is a simple token reference (`WORD`, in our case). Therefore, rule `a` says, ‘‘match the token identified as `WORD` on the input stream’’. The C function generated for rule `a` asks the lexical analyzer, generated by PCCTS, to collect characters until it sees a complete token. Each token in the vocabulary is given a unique number which the lexical analyzer returns to indicate what token was just matched. Function `a()` then verifies that the number associated with `WORD` is indeed returned by the lexical analyzer.

The above example can be tested via the following sequence of commands:

```
antlr -gk t.g
dlg -i parser.dlg scan.c
cc -I../h -o t t.c scan.c err.c
```

The first command generates the parser, `t.c`, the lexical description, `parser.dlg`, and a support file, `err.c`. The second command converts the lexical description to a C file that constitutes our scanner (lexical analyzer). The third command compiles all C files needed to generate the executable (the `-I../h` option tells the C compiler where to look for the standard PCCTS include files; you will have to change this to where the PCCTS include files are located). The output on our UNIX system looks like this (assuming the example is in file `t.g`):

```
% antlr -gk t.g
Antlr parser generator   Version 1.06   1989-1992
% dlg -i parser.dlg scan.c
dlg  Version 1.0   1989, 1990, 1991
% cc -I../h -o t t.c scan.c err.c
```

To test the grammar file, run the executable:

```
% t
hello
%
```

No error message is generated and `t` terminates successfully. If a token not in the vocabulary of our language is typed, an error message appears. We have only one word in our vocabulary, and hence, anything other than ‘‘hello, world’’ generates an error.

```
% t
bob
invalid token near line 1 (text was ’b’)
invalid token near line 1 (text was ’o’)
invalid token near line 1 (text was ’b’)
invalid token near line 1 (text was ’
ˆDline 1: syntax error at "EOF" missing WORD
%
```

The first ‘‘invalid token’’ errors are from the scanner, the last message is from the parser (function `a()`) indicating that end-of-file was found when a `WORD` was expected. `EOF` was returned by the scanner because `bob` was ignored and end-of-file appeared immediately afterwards; `EOF` is a predefined token in any PCCTS vocabulary.

Adding more tokens to your language’s vocabulary is easy—simply add more `#token` definitions. Consider this new example:

```
#token    "\ "     <<zzskip();>>            /* ignore blanks */
#token    "\t"     <<zzskip();>>            /* ignore tabs */
#token    "\n"     <<zzline++; zzskip();>>  /* ignore newlines */
#token A  "apple"
#token P  "pear"
```

This example introduces lexical actions—actions that are executed upon recognition of a particular regular expression. For most language descriptions, lexical actions are not used except to tell the scanner to skip a token or continue looking for more characters. `zzskip()` is a standard PCCTS function (generally, PCCTS variables/functions/defines are prefixed with `zz` to avoid name collisions with user variables) which forces the scanner to ignore the currently matched token and to try to find another. Essentially, the first three token definitions here tell the scanner that it is to ignore white space, but to increment the current line number when it sees a newline. The fourth and fifth definitions introduce two words into our vocabulary. Notice that only the last two have labels associated with them. Any `#token` instruction may give a label, but they are not necessary. Labels are handy when you want an action to refer to the value (token number) of a particular token; also, when a regular expression is complicated or confusing, often it is better to use a label throughout your grammar rather than repeating the regular expression. To illustrate this, we present the following four equivalent partial PCCTS descriptions:

(i) Repeated use of labels.

```
#token A "apple"
#token P "pear"

a : A P
  | P A
  ;
```

(ii) Repeated use of expressions.

```
#token "apple"
#token "pear"

a : "apple" "pear"
  | "pear" "apple"
  ;
```

(iii) Repeated use of implicitly-defined expressions.

```
a : "apple" "pear"
  | "pear" "apple"
  ;
```

(iv) Mixed use of labels and expressions.

```
#token A  "apple"
#token P  "pear"

a : "apple" P
  | "pear" A
  ;
```

Each unique token regular-expression string in PCCTS gets its own token number. Token labels are words that begin with a uppercase letter whereas rules begin with lower-case letters. Repeating the same token string in a grammar merely refers to the same token; strings can only appear once in `#token` definitions, however, as this instruction attempts to define a new token. An implicitly-defined token is one that is referenced but that has no formal `#token` instruction. In fact, we use the `#token` only when the expression is long, when a lexical action must be attached, or when a label is required (so that a C action can refer to it).

Each rule `a` above indicates that either `apple` followed by `pear` is to be matched or `pear` followed by `apple` is to be matched.

Once again, let’s test this vocabulary description with a complete, executable example:

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

<<main() { ANTLR(a(), stdin); }>>

#token    "\ "     <<zzskip();>>            /* ignore blanks */
#token    "\t"     <<zzskip();>>            /* ignore tabs */
#token    "\n"     <<zzline++; zzskip();>>  /* ignore newlines */

a : "apple" "pear"
  | "pear" "apple"
  ;
```

To build the executable, we proceed as before:

```
% antlr -gk t.g
Antlr parser generator   Version 1.06   1989-1992
% dlg -i parser.dlg scan.c
dlg  Version 1.0   1989, 1990, 1991
% cc -g -I../h -o t t.c scan.c err.c
```

To test the example, type:

```
% t
apple
     pear
%
```

No error is reported due to the validity of the input. Note that the newline and the spaces were ignored because of the `zzskip()` actions associated with our token definitions for white space. To ensure that `t` is actually doing something useful, try:

```
% t
apple apple
line 2: syntax error at "apple" missing pear
^D%
```

PCCTS generates parsers that automatically report errors and try to resynchronize the parser; hence, in this case, a control-D (`^D`) is necessary to terminate the program because `t` is looking for another token with which to resynchronize. Because of the `zzline++` statement in the action for newline, the error is correctly reported on line 2.

The regular expressions used in the above examples are simple and do not use any of the *meta-characters* or regular expression operators. Before presenting a more realistic example, we illustrate the use of some useful regular expression meta-characters (for a complete description see PCCTS documentation):

`@` `EOF` character

`\t` tab character

`\n` newline character

`\`*c* character escape; used to obtain actual character for meta-characters

`(`*e*`)` keep expression *e* as an indivisible group

`[`*c*`]` match one character from list *c*

`[`*x*`-`*y*`]`
match one character from range *x* to *y*

`~[`*c*`]` match one character not in list *c*

`{`*e*`}` expression *e* is optional

e`*` match zero or more of *e*

e`+` match one or more of *e*

e|f match either expression *e* **or** *f*

Naturally, the above operators and meta-characters can be used in many combinations to produce very complicated expressions. To illustrate more complex expressions, we define the vocabulary of a calculator (ignoring white space for the moment).

```
#token NUM "[0-9]+"
#token VAR "[a-zA-Z][a-zA-Z0-9]*"
#token     "\("
#token     "\)"
#token     "\+"
#token     "\-"
#token     "\*"
#token     "/"
```

A number is defined as a sequence of one or more decimal digits. Variables begin with an upper or lowercase letter, but can otherwise contain digits as well; note that `*` is used rather than `+` for variables because `+` would force `VAR` to recognize at least two characters. This calculator has some tokens in its vocabulary that are identical to those of the regular expressions, so these must be escaped to tell the scanner to look for those actual characters. To create an executable, we form a grammar which accepts one of the words in the vocabulary:

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

<<main() { ANTLR(a(), stdin); }>>

#token     "\ "     <<zzskip();>>            /* ignore blanks */
#token     "\t"     <<zzskip();>>            /* ignore tabs */
#token     "\n"     <<zzline++; zzskip();>>  /* ignore newlines */


#token NUM "[0-9]+"
#token VAR "[a-zA-Z][a-zA-Z0-9]*"
#token     "\("
#token     "\)"
#token     "\+"
#token     "\-"
#token     "\*"
#token     "/"

a : NUM | VAR | "\(" | "\)" | "\+" | "\-" | "\*" | "/" ;
```

As before, we create the executable with (assuming the example is in `t.g`):

```
antlr -gk t.g
dlg -i parser.dlg scan.c
cc -g -I../h -o t t.c scan.c err.c
```

The executable, `t`, will recognize any one token from our vocabulary. The next section discusses how one employs rules to specify valid, structured sequences; i.e. how one defines the syntax of a language.

3. Syntactic Analysis

The syntax of a language is the grammatical structure which summarizes the set of valid phrases in that language. Because one cannot normally delineate all possible sentences, languages are described via a set of rules which obey the laws of a *meta-language*, which is literally a ‘‘language to describe languages’’ just as the syntax of regular expressions represents a language. This section describes the format of a PCCTS language description—the syntax of PCCTS rules and how they may be used to impose a structure upon a stream of input tokens.

The basic template used to build a grammar is:

`#header` *action*
action(s) and/or `#token` *definition(s)*
rule(s)
action(s) and/or `#token` *definition(s)*

To compile, all grammars must define a number of things inside the `#header` action; this instruction is not optional and must appear first in your file. The rest of the file is basically a sequence of user actions, token and rule definitions—except that actions, not contained within rules, must be placed before or after the rule definitions.

Rules have the basic form:

rule: $alternative_1$ | $alternative_2$ | ... | $alternative_n$;

where $alternative_i$ is a sequence of the following elements:

token
Match *token* on the input stream.

rule Visit *rule* and match whatever is specified.

action
Execute C *action*.

(a_1 | a_2 | ... | a_n)
Introduce a subrule—match one a_i.

{ a_1 | a_2 | ... | a_n }
Introduce an optional subrule; match one a_i or none.

(a_1 | a_2 | ... | a_n)*
Conditionally match any sequence of a_i’s.

(a_1 | a_2 | ... | a_n)+
Match any sequence of a_i’s.

Examples of rule definitions are:

```
w   :   WORD ("," WORD)*
    ;
```

where rule `w` matches a list of comma-separated `WORD`'s. The `("," WORD)*` construction says match zero or more `"," WORD` sequences. Consider,

```
st  :   "if" expr "then" st {"else" st} ";"
    |   WORD ":=" expr
    |    "begin" ( st ";" )+ "end"
    ;
```

where `expr` is some rule that matches an arithmetic expression. Rule `st` matches statements such as:

```
if expr1 then begin
  i := expr2;
  j := expr3;
end
else
  k := expr4;
```

The first alternative has an optional subrule that matches an `else`-clause if it exists. The third alternative matches one or more semicolon-delimited statements, which are enclosed in `begin` and `end`. Let's examine the description of a simple expression.

```
e   :   e1 ( ("\+" | "\-") e1 )*
    ;

e1  :   WORD
    |   INT
    ;
```

Rule `e` matches simple expressions with only plus and minus as operators; e.g. `a+3-b` or `a`. Note that we have nested the `("\+" | "\-")` subrule within the `(...)*` subrule.

Let's build a complete PCCTS language description by extending the expression example. Rules to handle multiplication and division will be added as well as token definitions to ignore white space etc...:

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

<<main() { ANTLR(calc(), stdin); }>>

#token      "[\ \t]"  <<zzskip();>>          /* ignore blanks, tabs */
#token      "\n"      <<zzline++;>>          /* ignore newlines */
#token INT "[0-9]+"
#token FLOAT "[0-9]+ {. [0-9]+}"

calc:   ( e "\n" )* "@"
    ;

e   :   e1 ( ("\+" | "\-") e1 )*
    ;

e1  :   e2 ( ("\*" | "/") e2 )*
    ;

e2  :   INT
    |   FLOAT
    ;
```

Note that newlines are no longer to be ignored, hence, the `zzskip()` function call has been removed from its lexical action. Our language is a set of expressions terminated by newlines followed by end-of-file (@ is a predefined lexical meta-symbol referring to end-of-file). Without actions, testing this grammar is uninteresting because no output is generated (unless, of course, an invalid expression is given). Therefore, let us place an action among the rule elements to generate some output. Augment rule `calc` as follows:

```
calc:   ( e "\n" <<printf("found expression\n");>> )* "@"
    ;
```

Essentially, we have added C code to print out a brief message after an expression-newline pair has been encountered. Create the executable, `t`, as before with:

```
antlr -gk t.g
dlg -i parser.dlg scan.c
cc -I../h -o t t.c scan.c err.c
```

Test the program via a few simple expressions:

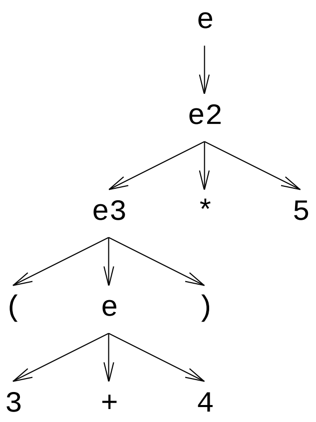
```
% t
3+4*5
found expression
3.15 / 6 - 2.1
found expression
^D%
```

This example grammar is not recursive; i.e. no rule references another rule that directly or indirectly returns to itself. But, recursion is a very powerful tool. It allows the concept of *self-similarity*. In other words, structures in which some subcomponents are similar to the outer structure. Pascal has several self-similar constructs: record field definitions, procedure definitions, expressions, and type definitions to name a few.

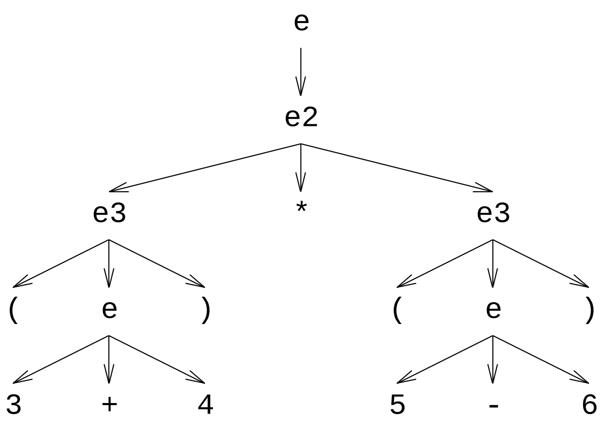
To illustrate recursive grammars, we extend the above expression example to allow parenthesized subexpressions such as `(3+4)*7`.

```
e2  :   INT
    |    FLOAT
    |   "\(" e "\)"
    ;
```

Placing the subexpression construct at the lowest recursion level makes it have the highest precedence because of the nature of top-down, depth-first parsing. To see this, consider the parse tree for `(3+4)*5` (beginning at rule `e`):



Clearly, `3+4` must be evaluated before the `*` for a valid result; this is precisely a depth-first evaluation of the parse tree (which PCCTS parsers do naturally). The deeper the recursive nesting, the higher the precedence. Extending the input expression to `(3+4)*(5-6)` yields:



Again, both operands of the `*` must be evaluated before it can proceed.

As another example of recursive definitions, consider type definitions for a Pascal-like language. Types look like:

```
char
integer
array [5] of char
array [100] of array [20] of integer
```

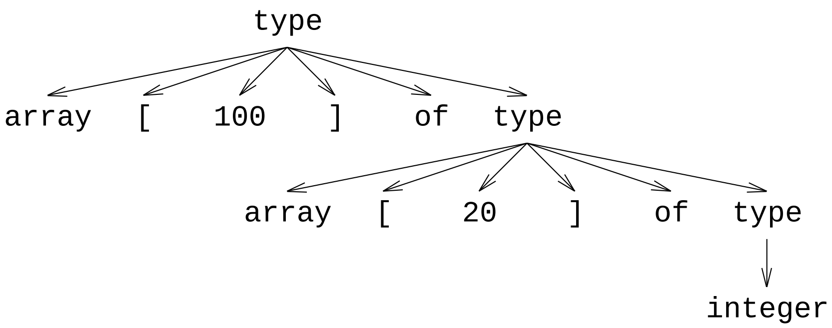
A grammar similar to the following could be used:

```
type:   "char"
    |   "integer"
    |   "array" "\[" INT "\]" "of" type
    ;
```

The recursive invocation of `type` by the `array` alternative effectively allows chains of `array` specifications. The parse tree for

```
array [100] of array [20] of integer
```

looks like:



In this case, we are less interested in precedence and more interested in allowing chains of array specifications.

In general, recursion and repetition constructs such as `(...)+` are needed to avoid delineating all possible phrases in a language. Grammars are descriptions of the patterns found among the phrases of a particular language just as Σ notation summarizes an infinite series.

The recognition of input languages, via the use of grammars, performs two tasks: it ensures phrase validity and directs translation to an output language. The next section demonstrates how actions, embedded among the grammar elements, can be used to effect a translation.

4. Translation

Given a grammar, PCCTS constructs a recognizer for phrases in that input language. No translation from input to output is performed. User actions must be supplied in the correct positions to generate output. Translation occurs when an action produces output which is a function of the input phrase. Actions have access to input phrase token values through an abstraction called an *attribute*. These attributes are user-defined types and can be as simple as the text associated with a token.

This section introduces the notion of an attribute as a means of communicating with the lexical analyzer and presents a number of examples that explain how and where actions can be used to generate output.

4.1. Attributes

Attributes are objects associated with all rules and rule elements, but we will only concern ourselves here with attributes associated with token and rule references. Attributes are referenced in actions with the notation `$`*i* where *i* indicates that the attribute for the i^{th} token in that production is desired. Attributes are run-time objects and have no value until run-time. They are generally used to access the actual text (or a function of the text) of the tokens matched on the input stream. The set of all tokens defines the *vocabulary* of the input language. The term ‘‘token’’ collectively refers to the token type (an integer that identifies it as part of the vocabulary) and the token text (the actual string that matched the regular expression for the token type).

Before illustrating attributes, we begin with an example. The vocabulary of an input language (known a priori) may be the set { `WORD`, `"begin"`, `INT` }, which is the set of integer token types. The text associated with a token type is only known at parser run-time because it depends on the input characters. Let us say that the grammatical structure of the language is any sequence of tokens in the vocabulary (ignoring white space); then, a valid sentence could be:

```
begin hello 34 13 bob
```

The parser would see a token stream of tuples of the form (token type, token text):

```
(begin, begin)
(WORD, hello)
(INT, 34)
(INT, 13)
(WORD, bob)
```

A different input sentence, with the same sequence of **token types** is:

```
begin hi 2 99 ptr
```

which would yield the same sequence of token types, but a different set of token text:

```
(begin, begin)
(WORD, hi)
(INT, 2)
(INT, 99)
(WORD, ford)
```

The grammar might look like:

```
a : ( WORD | "begin" | INT )+
  ;
```

Only the token types are referenced in the grammar as they describe the structure of the language and are a shorthand notation for the set of valid input sentences. Obviously, one could not delineate all possible sentences as there are infinitely many. For a PCCTS description to perform a translation that is specific to the particular input, actions must access the text of the input tokens, not just the token type. Attributes are provided to provide access to the text (or some function thereof) of an input token. To illustrate this, we give a complete example and then, later, describe the particulars:

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

<<main() { ANTLR(a(), stdin); }>>

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

a : ( WORD    <<printf(" %s", $1);>>
    | "begin" <<printf(" begin");>>
    | INT     <<printf(" %s", $1);>>
    )+
  ;

#token WORD "[a-z]+"
#token INT  "[0-9]+"
```

This example defines attributes to be strings representing what was found on the input stream and prints the stream of tokens back out. In other words, attributes are merely a copy of the words found; the mapping from token/lexeme to attribute is an identity mapping (do nothing but copy). For the moment, concentrate on the actions. `$1` refers to the attribute of the first item in the production in which the action occurs; in this case, only one item appears per production. Note that the action for the `"begin"` token does not need to refer to its attribute as it will always be `begin`. The rest of this section describes the particulars needed to understand everything else in the example.

PCCTS requires that the user define the data type or structure of the attributes as well as specify how to convert from lexemes to attributes. The type is always defined by a C `typedef` named `Attrib` and must be defined in the action associated with the `#header` instruction. For example, if one wishes the attribute for a token to be simple integers, the following is a sufficient type definition:

```
#header <<typedef int Attrib;>>
```

However, this does not tell PCCTS how to convert a token to an attribute. This is accomplished with a function called `zzcr_attr()` which defines the value of an attribute given complete information about a lexeme (token number and associated text). It has the general form:

```
void
zzcr_attr(a,token,text)
Attrib *a;
int token;
char *text;
{
    /* *a = function(token, text); */
}
```

where `a` points to an attribute created by PCCTS at run-time. The user simply has to assign a value to `*a`. In our case, we will use a macro version to set our attributes to the integer value of the input:

```
#define zzcr_attr(a,tok,txt) {*(a) = atoi(txt);}
```

This specifies that whenever a token is matched on the input stream by the parser, an attribute of type `int` is to be created and assigned the result of `atoi(`*text*`)` where *text* is the character string matched for the token. The attribute is then made available as `$`*i* to actions in the production that matched the token. For example,

```
#header <<
    typedef int Attrib;
    #define zzcr_attr(a,tok,txt) {*(a) = atoi(txt);}
>>

<<main() { ANTLR(a(), stdin); }>>

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

a   :   "hi" "[0-9]+" <<printf("$1, $2 are %d, %d\n", $1, $2);>>
    ;
```

`$1` refers to the first token in the alternative, `"hi"`; similarly, `$2` refers to the the second token, `"[0-9]+"`. When executed, the executable `t` (created as before) yields:

```
% t
hi 34
$1, $2 are 0, 34
%
```

where `atoi()` of a non-numeric string is 0, but the text `34` gets converted to an integer (binary word) version of 34 and printed back out as a number.

The token type can be tested to ensure that it is an integer before applying the `atoi()` function via:

```
#header <<
    typedef int Attrib;
    #define zzcr_attr(a,tok,txt) {if ( tok==INT ) *(a) = atoi(txt);}
>>
```

where `INT` is defined to be `"[0-9]+"`. This defines an attribute for all `INT` tokens found on the input stream. Other tokens have undefined attributes.

Attributes can have multiple elements or assume one of many values. For example, we can extend the above example to handle `FLOAT` tokens as well:

```
#header <<typedef union { int ival; float fval; } Attrib;>>

<<
void
zzcr_attr(a,token,text)
Attrib *a;
int token;
char *text;
{
    switch ( token )
    {
        case INT   : (a)->ival = atoi(text); break;
        case FLOAT : (a)->fval = atof(text); break;
    }
}
>>
```

The `typedef` specifies that attributes are integer or floating point values. When the regular expression for a floating point number (integer number) is matched on the input stream, `zzcr_attr()` converts the string of characters representing that number to a C `float` (`int`).

Attributes can become even more complicated, but typically, attributes are merely a copy of the text found on the input stream. A standard PCCTS attribute definition is available as `charbuf.h` and is defined as follows:

```
/* PCCTS attribute -- constant width text */
#ifndef D_TextSize
#define D_TextSize   30
#endif

typedef struct { char text[D_TextSize]; } Attrib;

#define zzcr_attr(a,tok,t)   strncpy((a)->text, t, D_TextSize-1);
```

These attributes are referred to by `$`*i*`.text` in actions.

Each alternative begins a new sequence of `$`*i*'s and from an enclosing scope/level, entire subules are counted as one unit. This is best explained with an example:

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

From an action after token `E`, `A` is `$1`, `B` is `$2`, the entire subrule `( C D )` is `$3`, and `E` is `$4`; `C` and `D` are inaccessible from outside the scope of the subrule. From an action inside the subrule just after the token `D`, `C` is `$1` and `D` is `$2`. In alternative two from an action after `G`, `F` is `$1` and `F` is `$2`. Attributes have a scoping just like variables in a programming langauage.

Attributes are a means of communicating with the lexical analyzer. Actions may use these attributes to provide a translation. The next section utilizes the concepts presented here to build translators.

4.2. Actions

Actions are rule elements just like token references, but perform a different function. Token references indicate that a particular token is to be matched on the input stream at that point in the parse. Actions indicate that this action is to be performed at that point in the parse, immediately following the preceding token match. For example,

```
a    :    A <<action1>> B ;
     |  ( C )+ <<action2>>
     ;
```

$action_1$ is executed after the parser has found an `A`, but before it has found a `B`. $action_2$ is executed only after a sequence of one or more `C`'s has been found.

As a more concrete example, we augment the above `calc` example to print something more useful than `found expression`:

```
calc:    ( e "\n" <<printf("\n");>> )* "@"
     ;

e    :    e1
          (    (   "\+" <<printf(" add");>>
               |  "\-" <<printf(" sub");>>
               )
               e1
          )*
     ;


e1   :    e2
          (    (   "\*" <<printf(" mult");>>
               |  "/"   <<printf(" div");>>
               )
               e2
          )*
     ;

e2   :    INT      <<printf(" INT");>>
     |    FLOAT    <<printf(" FLOAT");>>
     ;
```

Essentially, we have added C code to print out the operand types and operators. Create the executable, `t`, as before with

```
antlr -gk t.g
dlg -i parser.dlg scan.c
cc -I../h -o t t.c scan.c err.c
```

Test the program via a few simple expressions:

```
% t
3+4*5
 INT add INT mult INT
3.15 / 6 - 2.1
 FLOAT div INT sub FLOAT
ˆD%
```

Now, let’s use the attributes to generate code for a simple *reverse-polish* stack machine whose operations are defined as follows:

`push` *opnd*

  Push *opnd* onto the stack.

`print`

  Print the value of the top of stack; POP the value off the stack.

`add`

  PUSH(POP + POP)

`sub`

  a := POP

  b := POP

  PUSH(*b* - *a*)

`mult`

  PUSH(POP * POP)

`div`

  a := POP

  b := POP

  PUSH(*b* / *a*)

Modify the rules as follows:

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

<<main() { ANTLR(calc(), stdin); }>>

#token     "[\ \t]"  <<zzskip();>>           /* ignore blanks, tabs */
#token     "\n"      <<zzline++;>>           /* ignore newlines */
#token INT "[0-9]+"
#token FLOAT "[0-9]+ {. [0-9]+}"
```

```
calc:   ( e "\n" <<printf("\tprint\n");>> )* "@"
    ;

e   :   <<char *op;>>
        e1
        (   (  "\+" <<op="\tadd\n";>>
            |  "\-" <<op="\tsub\n";>>
            )
            e1
             <<printf("%s", op);>>
        )*
    ;


e1  :   <<char *op;>>
        e2
        (   (  "\*"  <<op="\tmult\n";>>
            |  "/"   <<op="\tdiv\n";>>
            )
            e2
             <<printf("%s", op);>>
        )*
    ;

e2  :   INT     <<printf("\tpush %s\n", $1.text);>>
    |   FLOAT   <<printf("\tpush %s\n", $1.text);>>
    ;
```

Table of Contents