Designing a Basic Block Profiler

Last Update Time-stamp: "97/07/04 01:20:59 zdu"

This note serves as a advanced tutorial on the use of lex and yacc and how to control the interactions between them. It documents the design of a source-to-source transformation tool which enables profiling the basic blocks of Ansi-C programs. The tool transforms a C source file into another *instrumented* source file. When the instrumented source file is compiled and executed, it produces a count of the number of times each of its basic blocks is executed.

Requirements

A *basic block* is a maximal sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or the possibility of branching except at the end. Basic blocks can be used to capture the control-flow information associated with a particular execution of a program. Specifically, we define the *basic blocks* of a C program to include the entry points to all functions, as well as all points indicated by a ^ in the following constructs (recursively expanding `statement`s):

```
if (expr) statement
          ^          ^

if (expr) statement else statement
          ^               ^          ^

while (expr) statement
             ^          ^

for (expr; expr; expr) statement
                       ^          ^

do statement while (expr)
  ^                        ^

label: statement
       ^

case const-expr: statement
                 ^

default: statement
         ^
expr ? expr : expr
       ^      ^
```

For each basic block in the program being profiled, the profiler is required to produce a count of the number of times execution flows through that basic block.

Overall Design

The profiling tool operates in three phases:

1. **Instrumentation Phase**: The tool modifies the source program so as to enable it to produce its basic block profile when it is subsequently executed.
2. **Execution Phase**: The instrumented program is run to produce its basic block profile.
3. **Post Execution Phase**: The user browses and collates the basic block profile using general tools and scripts.

Basic block profiling is achieved by associating a unique counter with each basic block. During the instrumentation phase, code is inserted at each basic block to increment its associated counter. Code is also inserted to ensure that the counts in **all** basic block counters are captured in a file when the program exits.

The instrumentation phase assumes that the source program is syntactically and semantically correct. Hence it may do minimal error-checking.

The instrumentation phase is implemented using three main components:

a) A standard C preprocessor. Any off-the-shelf preprocessor can be used. This component may be omitted if requested via a command-line option.
b) A C scanner.
c) A C parser which understands C well enough to produce the instrumentation.

The input to the *scanner* is fed by the output of the C preprocessor. When requested by the parser, the scanner tokenizes its input and passes the next token onto the parser. It maintains the name of the current source file and the current line number (obeying any `#` directives from the preprocessor specifying the current filename and line number). Along with each token, the scanner passes any *whitespace* between the previous token and the current token, where *whitespace* includes the text of comments and residual preprocessor directives. The scanner is implemented using a version of `lex`.

The *parser* parses the token stream passed to it by the scanner. As it parses each input token, it emits it's associated text (including whitespace) to the output stream associated with the instrumented source. When it recognizes a basic block it emits suitable instrumenting code to the output stream. Note that since C preprocessors do not concatenate adjacent string tokens, the parser treats adjacent string tokens as though they had been concatenated. The parser is implemented using a version of `yacc`.

This design appears somewhat clumsy because of the passing of the text of all preceeding whitespace, comments and preprocessor tokens from the scanner to the parser. An alternative design would make the scanner responsible for copying the source program to the output stream. All that the parser would need to emit would be the additional code required to instrument the source program. In fact, an earlier version of this program did use such a design, but it was abandoned for the following reason:

> In some situations, the parser requires a lookahead token before it can execute any actions which emit the additional instrumenting code; in other situations it does not. Unfortunately, the requests made to the scanner do not distinguish between a request for a lookahead token versus a request for a token which will be shifted immediately. Hence it becomes difficult to synchronize the output actions of the

scanner with the output actions of the parser to ensure that the instrumenting code is inserted at precisely the right point in the source file. It can be done (using features like the `%nolook` directive provided by zyacc), but the resulting parser can be quite fragile.

Instrumenting a Program

The inserted instrumenting code has to achieve the following:

1. Declare and define all the auxiliary entities needed for profiling.
2. Increment the unique counter associated with a basic block whenever that basic block is executed.
3. Ensure that all profiling information is captured in persistent storage before the program exits.

(1) is easy to achieve by including declarations and definitions in the instrumented file. The entities include an array of counters (one counter for each basic block), an array of line numbers (one line number for each counter), and an array of file names (one file name for each counter, at least conceptually), and an entity which points to all the other entities and contains other miscellaneous information. To facilitate instrumentation using a single pass, the entities are first declared at the beginning of the instrumented file and defined only at the end of the file when their sizes are known.

(2) is trivial to achieve: a simple statement which increments the appropriate member of the counter array is inserted. However, we must be careful as we insert the counter increments. Consider the following `if` statement:

```
if (a != 0) b= c/a;
```

Assuming the `then` basic block controls counter `_BBCntrs[235]`, then we could insert code `_BBCntrs[235]++` just before the `then`-statement as follows:

```
if (a != 0) _BBCntrs[235]++; b= c/a;
```

Unfortunately, we have changed the user's semantics of the program --- the statement `b= c/a` is no longer governed by the `if`-test!

The correct way to do this is to insert the counter increment surrounded by braces as in:

```
if (a != 0) { _BBCntrs[235]++; b= c/a; }
```

Though braces are not always necessary when inserting the counter increment before a statement, it is simplest to merely insert them in all cases.

Similarly, when incrementing the counters for a conditional expression like

```
max= (a > b) ? a : b;
```

we insert the counter incrementing code using the comma-operator:

```
max= (a > b) ? ( _BBCntrs[441]++ , a ) : ( _BBCntrs[442]++ , b );
```

In this case, the inserted parentheses are always necessary.

(3) can be achieved by registering a function to be called when the program exits. This can be done using C's `atexit()` library function. Unfortunately, in the absence of any information about the global control flow of the program being instrumented, there is no really good place to call `atexit()`. The solution used is to insert code at the beginning of each function in a file to check whether `atexit()` has been called for that file; if it has not been called, then it is called to register a function specific to that file.

The Scanner

The scanner in scan.l is a straight-forward use of `lex` technology. Multi-character tokens involving non-alphanumeric characters like <<= and `...` are recognized by merely listing out the patterns describing them. Single character non-alphanumeric tokens like < and = are recognized using a catch-all `.` pattern.

Reserved words like `for` and `switch` are treated just like identifiers as far as the patterns are concerned. During initialization, the reserved words are entered into a *string-table* (maintained by an auxiliary library module). The string table assigns successive indexes or `ID`s (starting at 0) to each distinct identifier it is given. When a identifier is recognized via a `lex` pattern, it is looked up in the string-table. If its `ID` is less than the number of reserved words, then it corresponds to a reserved word; its token number is computed using a simple computation. Otherwise it must be a C identifier and a `ID_TOK` token is returned.

The only complicated patterns are those for floating point numbers, strings and character constants. These are simplified by using auxiliary `lex` *macros* which are defined in section 1 of the `lex` file. Since we are not concerned about the semantics of most tokens, we do not process escape sequences in any detail greater than needed to find the end of a string or character constant.

Comments are handled using an exclusive start state. In the `COMMENT` start state, comment lines which do not contain `*`s, are processed a line at a time to maximize performance. There is a special case pattern to handle a sequence of `*`s which occur within the comment but do not terminate the comment (they are not immediately followed by a `/`). There is another special case pattern to handle `'\n'`s within a comment.

Line numbers are tracked using zlex's yylineno feature.

The scanner is written so that it can be used to scan C text which may or may not have been previously preprocessed (depending on a command-line option). Preprocessor directives are handled by another exclusive start state `CPP` which is entered when a `#` is seen at the beginning of a line. It uses an auxiliary state maintained in a global C variable to record whether it has seen something which looks like a `#line` directive or a `#` followed by a line number and a file name. It it finds such a directive, it records the line number and file name in a global variable. It ignores all other characters until it sees a `'\n'` at which point it returns to the `INITIAL` start state. It is important to note that since the number mentioned in the `#` directive is the number of the *next* line, the `yylineno` variable is updated to one less than that specified so that when it is incremented after seeing the terminating `'\n'`, it will be correct.

Passing the text of each token (including preceeding whitespace) is handled using two dynamically-grown buffers. Two buffers are sufficient to handle upto a single token of lookahead by the parser. The buffers may get quite large when large comments are encountered, but most existing computer systems should be able to cope with the sizes.

The parser's interface to the scanner does not directly use the `lex` generated `yylex()`. Instead, the interface is a wrapper `scan()` around `yylex()`. The wrapper is responsible for the following tasks:

a) It takes care of the necessary cleanup necessary when the scanner has hit the end of its current file.
b) It concatenates the lexeme (contained in `yytext`) of the token being returned to the current buffer. (The lexemes for tokens like whitespace and comments which are not returned to the parser, are added directly to the buffer within the action associated with the corresponding pattern).
c) It extracts the text of the current buffer into the semantics (`yylval`) associated with the current token and switches buffers in preparation for the next token.

The current filename is intern'd and stored in the string-table module. The filename and line number are maintained as part of `yylval` --- the semantic value of the token. This has the advantage that the parser can access the line number of a token which was scanned much earlier, but has the disadvantage that the size of parser stack entries may be increased.

The semantic value of a token also contains the buffered text of the current lexeme including its preceeding whitespace, plus the total length of the buffered text as well as the length of the preceeding whitespace.

The other field which is returned as part of the `yylval` semantics of a token is the string-table `ID` of any identifier token. The parser needs this field to distinguish between identifiers used as `typedef`-names and those used for other purposes.

The scanner also provides a routine the parser can use to start scanning a new file. Depending on a command-line option, the routine uses the C preprocessor to first preprocess the new file into a temporary file (this is more portable than using a *pipe*). It tries to read the name of the preprocessor to be used from variables in the environment.

The Parser

The parser in parse.y uses the grammar given in the ANSI-C standard fairly directly. The grammar is organized in the order given in K&R2 rather than the order used in the actual ANSI standard, as that order appears preferable. Except for the `typedef-name` non-terminal, the standard grammar is suitable for LALR(1) parsing after productions containing optional constructs are duplicated, once with and once without the optional construct. Since `typedef-name`s cannot be recognized purely syntactically, the `typedef-name` nonterminal is recognized using a semantic predicate.

The grammar is divided into the following sections:

Declarations
: This section takes care of entering `typedef-name`s into a simple symbol table when they are declared, and recognizing identifiers as `typedef-name`s when they are referenced.

Statements
: This section takes care of emitting the additional instrumenting code necessary to update profiling counters at the start of each basic block.

Expressions
: This section contains the expression grammar of the ANSI-C standard largely unchanged except for the addition of actions for profiling `?:` conditional expressions.

Terminals
: Each actual terminal is accessed via a nonterminal (which is treated as a pseudo-terminal in the rest of the grammar). There is a pseudo-terminal for each real terminal, where each pseudo- terminal is responsible for copying its lexeme (including preceeding whitespace) to the output stream. All output (including keeping track of the counters) is done using out.c (with declarations in out.h).

Typedef Names

One of the challenges involved in parsing C is that parsing decisions depend on whether or not a particular identifier is a `typedef-name`. The recognition of `typedef-name`s cannot be achieved using purely syntactic methods but requires some semantic knowledge. One possible solution is to make the scanner lookup an identifier in a symbol table and return a special `TYPEDEF_NAME` token if it is a `typedef-name`. Unfortunately, such a solution can be quite difficult to implement because even when an identifier has been declared to be a `typedef-name`, the scanner should still return it as a normal `ID` identifier in contexts where the identifier is being redeclared. What is really needed is a way for the semantics of a token to affect the parsing decisions made by the parser: the zyacc parser generator provides such a mechanism via semantic predicates.

The overall strategy used to process `typedef-name`s is as follows:

> When a `typedef` declaration is parsed, all the identifiers declared in it are entered into a scoped *typedef table*. Then when the parser encounters a reference to an identifier in a context where a `typedef-name` is possible, it uses a semantic predicate to check whether the identifier is indeed a `typedef-name` by looking it up in the typedef table.

The correct processing of a `typedef` declaration is achieved by associating a synthesized attribute `$dclType` with the `declaration_specifiers` nonterminal. This attribute is set to `TYPEDEF_DCL` if the `typedef` `storage_class_specifier` is specified. The other possibilities for this attribute are a `ID_DCL` if a non-typedef identifier is declared in the same namespace as typedef-names, and a `OTHER_DCL` for an identifier in another namespace (like structures). This synthesized attribute is propagated via inherited attributes to `init_declarator_list`, `init_declarator`, `declarator` and `direct_declarator`. The base production (`ID`) for `direct_declarator` checks if this attribute is not `OTHER_DCL`; if so, it adds the `ID` to a simple symbol table maintained for the `typedef` namespace, specifying whether or not it is a typedef. (Since `declaration_specifiers` is a simple linear list of `storage_class_specifier`s, `type_specifier`s and `type_qualifier`s, it is possible to have propagated the `dclType` information using a single global attribute rather than the multiple synthesized and inherited attributes used above).

When a `typedef` declaration is recognized, the `ID` of the identifier being declared is added to a scoped symbol table. The table uses a simple hashtable search. A new scope is begun at the beginning of each compound statement and the current scope is closed at the end of each compound statement; this is achieved by actions associated with the nonterminals `lbrace` and `rbrace`. Collision resolution is achieved by chaining, with the chained symbols maintained in a stack. Ending a scope is achieved by

merely popping this stack, after ensuring that all the popped symbols are removed off their hash-chains.

What remains is to handle the situation where a existing `typedef-name` is referenced. This is complicated by the fact that a `typedef-name` may be redeclared; fortunately, the standard requires that any such redeclaration have at least one `type_specifier` among its `declaration_specifiers`. Hence an identifier occurring in a declaration is recognized as a `typedef-name` only if no `type-specifiers` have been seen previously in the declaration --- this is tracked by means of a global attribute `seenType`. Hence a identifier should be recognized as a `typedef_name` only if both the following conditions are true:

- `seenType` is `FALSE`.
- The identifier is in the current typedef table as a `typedef`.

This test is done using the `%test` semantic predicate: note that the test is performed on the lookahead token.

Instrumenting Code

The statements section of the grammar is responsible for inserting the instrumenting code. This is achieved by associating `CounterType` inherited and synthesized attributes with each `statement` nonterminal, where `CounterType` is defined as follows:

```
typedef enum {
  NO_COUNTER,           /* no counter needed */
  COMPOUND_COUNTER,     /* counter at start of compound */
  COND_COUNTER,         /* counter for ?: conditional expressions */
  FN_COUNTER,           /* counter for function entry */
  STMT_COUNTER          /* counter for statements */
} CounterType;
```

A `FN_COUNTER` differs from a `STMT_COUNTER` in that it also inserts code to ensure that `atexit()` has been called. A `COMPOUND_COUNTER` differs from a `STMT_COUNTER` in that it is safe to insert the incrementing code for a compound counter directly within a `compound_statement`, whereas code for a `STMT_COUNTER` can only be safely inserted if paired with the corresponding `statement` within braces.

The inherited attribute of `statement` denotes the counter type needed just before that `statement`; the synthesized attribute denotes the counter which will be needed for the *next* statement. All the productions for `statement` except the one for `compound_statement` use the inherited attribute to insert a left brace followed by the appropriate counter increment (the action is encapsulated in the nonterminal `counter_begin`), and finish up by inserting a closing right brace. The inherited attribute is simply passed down to `compound_statement`.

Most of the nonterminals for the individual kinds of statements merely have a synthesized attribute which specifies the kind of counter needed by the *next* statement. However, in many situations the *statement* nonterminal does not need to look at this attribute:

- An `expression_statement` will never need to be followed by a counter increment. Hence `expression_statement`s do not have any `CounterType` attributes; the `expression_statement` rule within `statement` asserts this fact by setting the attribute `$counterZ` of `statement` to `NO_COUNTER`.
- A `selection_statement` or an `iteration_statement` must always be followed by a counter increment. Hence the corresponding rules in `statement` ensure this by always setting the attribute `$counterZ` of `statement` to `STMT_COUNTER`.
- Surprisingly enough, the `jump_statement` rule in `statement` also specifies attribute `$counterZ` to be `NO_COUNTER`. However, this makes sense since a jump statement represents an unconditional control transfer and the statement following the jump statement will never be executed unless control is transferred to it in some way other than sequentially from the jump statement. In that case, the counter increment code will be inserted there by some other rule.

A counter is required after a `labelled_statement` or a `compound_statement` depending on each individual statement. Hence in those cases, the `$counterZ` attribute of `statement` is set to the value returned by the corresponding nonterminal.

The computation of the attribute `$counterZ` in the rules for each individual kind of statement are straight-forward. This attribute should be set to the type of counter needed by the next statement. For example, a counter is needed after a if-then-else statement if it is needed after the then-statement or it is needed after the else-statement.

Note that many of the newly introduced empty nonterminals are necessary for avoiding conflicts. For example, if the `counter_begin` nonterminal in the rules for `statement` was directly replaced by the corresponding action, conflicts would result. That is because even though the actions are identical, the parser generator does not realize it and creates different empty rules for each insertion of a mid-rule action, resulting in reduce-reduce conflicts. Similarly, the `needs_counter_statement` nonterminal in the rules for `selection_statement` is necessary to avoid an attribute conflict, as the parser generator does not realize that the value `STMT_COUNTER` being assigned to the inherited attribute of `statement` is the same in both cases.

Execution Phase

The instrumented program must be linked with the zprof library before it can be executed. The library provides a routine `_bbProfOut()` which is called by each program file on exit to append basic block counts to the file `zprof.out`.

Post-Execution Phase

The format of the file `zprof.out` consists of lines of the form:

filename:*linenum*: *count*

specifying that line number *linenum* in file *filename* was executed *count* times. If there are multiple basic blocks associated with a particular source line, then multiple lines will be produced for the same *filename* and *linenum*.

The above format is amenable to easy processing by several tools:

- The file can easily be sorted in descending order of counts:

```
sort -t: -k3 -nr zprof.out
```

- Emacs compilation-mode can be used to interactively browse this file. `M-x compile` can be used to start compilation and the compilation command can be specified simply as `cat zprof.out`. Then the `M-`` command can be used repeatedly to position the cursor at each source line with its execution count shown in the compilation buffer.
- The trivial Perl script `zcounts` distributed with this program can be used to generate annotated versions of each source file (with a `.bb` extension) where each source line is preceeded by its execution count.

Feedback: Please email any feedback to zdu@acm.org.

Back to zprof home page