PCCTS Version 1.0x Advanced Tutorial

Terence Parr, Hank Dietz, Will Cohen

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
Spring 1992
`parrt@ecn.purdue.edu`
`hankd@ecn.purdue.edu`
`cohenw@ecn.purdue.edu`

Constructing a translator can be viewed as an iterative refinement process moving from language recognition to intermediate-form transformation. This document presents one possible sequence of refinements. We shall use as many features of PCCTS as is reasonable without regards to optimality.

We will develop a compiler for a simple string manipulation language called *sc*. The compiler will generate code for a simple stack machine.

This document assumes familiarity with the concept of a parser generator and other language recognition tools — in particular, a passing familiarity with YACC is useful.

The development will proceed as follows:

(i) Define a grammar to recognize functions, statements, variable definitions and expressions in *sc*.

(ii) Add symbol table management to the grammar developed in *(i)*

(iii) Add actions to translate *sc* into stack code for a simple stack machine.

(iv) Construct trees as an intermediate-form.

(v) Build a code-generator to translate the trees into executable C code.

1. *sc* Language definition

sc is a simple, C-like language with only one data object type— string. It has global variables, functions, arguments and local variables. There are no arrays or structures.

1.1. *sc* Expressions

Expression atoms are limited to variables, string constants (`"`*character(s)*`"`) and function calls (`function(`*expr*`)`). The string constants `"false"` or `"0"` represent false and `"true"` or `"1"` represent true. A limited set of operators are available (from highest to lowest precedence)

`-` Unary operator negate. Numeric operator.

`*, /` Binary operators multiply and divide. Groups left-to-right. Numeric operator.

`+, -` Binary operators add and subtract. Groups left-to-right. Numeric operator except for `+` which is ‘‘concatenate’’ if one of operands is non-numeric.

`==, !=` Binary operators equal, not equal. Groups left-to-right. Numeric or non-numeric.

`=` Binary assignment operator. Groups right-to-left. For example, `a = b = c` means to assign `c` to `b` and then to `a`.

1.2. *sc* Functions and statements

Functions may have multiple local variables, but at most one parameter. All functions implicitly return a string to the calling function, although the value need not be used. If a `return`-statement is not executed within a function, the return value for that function is undefined. Functions are defined exactly as in C:

```
function(arg)
{
    statement(s)
}
```

A function called `main()` must exist so that our interpreter and compiled code knows where to begin execution. Also, `main()` always has a implicitly passed parameter which contains any command-line argument.

sc has six statements.

`if` Conditionals have the form:

```
if ( expr ) statement
```

`while`

Loops have the form:

```
while ( expr ) statement
```

`{` *statement(s)* `}`
: Block statements treat more than one statement as a single statement as in C. However, one cannot define variables local to a block statement.

expr`;` An *expr* can include any valid *sc* expression, but only expressions involving assignments and function calls are useful.

`return` *expr* `;`
: This statement behaves exactly like the C return statement except that only strings can be returned.

`print` *expr*`;`
: Print the string described by *expr* to `stdout`.

1.3. Example

The following code is a simple *sc* program that computes *n* factorial.

```
main(n)
{
    print fact(n);
    print "\n";
}

fact(n)
{
    if ( n == "0" ) return "1";
    return n * fact(n - "1");
}
```

2. Language recognition

We begin our project by constructing a parser—a program that will recognize phrases in *sc*. The first two subsections describe the *sc* lexicon while the later subsections present the *sc* grammar with and without symbol table management.

2.1. Lexical analysis

Our language has only strings of upper- and lower-case letters, called `WORD`'s, operators, string constants and a few grammatical grouping symbols; all of which except `WORD` will be defined implicitly by mentioning them in the grammar. `WORD` will be placed last in the grammar—i.e. after all keywords have been defined. Since keywords are really just strings of characters as well, they must be treated specially. When two (or more) regular expressions can be matched for the current input text, DLG scanners resolve the ambiguity by matching the input to the expression mentioned first in the description. Hence, we place

```
#token WORD "[a-zA-Z]+"
```

at the bottom of the file.

Tabs, blanks and carriage-returns are all considered white space and are to be ignored (although we wish to track line numbers). The following ANTLR code will instruct the parser to skip over white space.

```
#token "[\t\ ]+"    << zzskip(); >>         /* Ignore White */
#token "\n"         << zzline++; zzskip(); >>
```

Strings are strings of characters enclosed in double-quotes. For simplicity we will allow any character not in the ASCII range 0..0x1F (hex) plus any "escaped" version of the same character set.

```
#token STRING       "\"(~[\0-\0x1f\"\\]|(\\~[\0-\0x1f]))*\"" <<;>>
```

2.2. Attributes

The DLG scanner matches keywords, `WORD`'s, etc... on the input stream. The way ANTLR parsers access that text is though the attribute mechanism. Attributes are run-time objects associated will all grammar elements (items to the right of the `:` in a rule definition). Although attributes are associated with subrules and rule references, only those attributes linked to grammar tokens are of interest to us. Attributes are denoted `$`*i* where *i* is the grammar element *i* positions from the start of the alternative. The user must define an attribute type called `Attrib`. This type is most often some sort of string whether constant or dynamically allocated. For simplicity, we employ a standard attribute type available with the PCCTS system. Attributes are structures that contain a constant width (30 character) string. By placing the character array inside of a structure, the entire string can be copied like any simple C variable. The ANTLR pseudo-op `#header` is used to describe attributes and other information that must be present in all generated C files. In our case, we simply need to include the standard text attribute definition. This pseudo-op must occur first in the grammar file if it exists at all.

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

2.3. Grammatical analysis

This subsection describes the grammatical or syntactic aspects of *sc*. No symbol table management is used and therefore functions and variables are considered simple `WORD`'s. Later versions of the grammar can use the tokens `VAR` and `FUNC`.

2.3.1. Definitions, variables

An *sc* program is a sequence of definitions—variables or functions:

```
p       :   ( func | "var" def ";" )* ;
```

The `( )*` subrule means that there are zero or more definitions. The `|` operator starts a new alternative.

The grammar for a variable definitions is broken up between the rule `p` and `def` so that `def` can be reused for parameter definitions (it also makes more sense when code generation has been added).

```
def     :   WORD ;
```

2.3.2. Functions

sc functions follow C's format except that the default return type is a string instead of `int`.

```
func    :   WORD "\(" { WORD } "\)"
            "\{"
                ( def )*
                ( statement )*
            "\}"
        ;
```

Note that the parentheses and the curly-braces must be escaped with `\` because they are special regular expression symbols.

2.3.3. Statements

The statements outlined in the *sc* language definition can be described with

```
statement
        :   expr ";"
        |   "\{" ( statement )* "\}"
        |   "if" "\(" expr "\)" statement {"else" statement}
        |   "while" "\(" expr "\)" statement
        |   "return" expr ";"
        |   "print" expr ";"
        ;
```

where `{ }` means optional.

2.3.4. Expressions

The operators and expression atoms described in the language definition can be recognized by the following grammar fragment.

```
expr    :   WORD "=" expr
        |   expr0
        ;

expr0   :   expr1 ( ("=="|"!=") expr1 )*
        ;

expr1   :   expr2 ( ("\+"|"\-") expr2 )*
        ;

expr2   :   expr3 ( ("\*"|"/") expr3 )*
        ;

expr3   :   {"\-"} expr4
        ;

expr4   :   STRING
        |   WORD { "\(" { expr } "\)" }
        |   "\(" expr "\)"
        ;
```

Rule `expr` is ambiguous if we only look one token into the future since `WORD` can also be an `expr0`. However, if we were to tell ANTLR to use two tokens, it could see that the `"="` assignment operator uniquely identifies which alternative to match when `WORD` is the first token of lookahead. Rule `expr` makes this grammar LL(2); but, we only use the extra token of lookahead in `expr` (leaving all others LL(1)). Please note the use of ANTLR's command-line option ‘‘`-k 2`’’ in the makefiles presented below.

ANTLR does not have a method of explicitly outlining operator precedence. Instead precedence is implicitly defined by the rule invocation sequence or abstractly by the parse-tree. Rule `expr` calls `expr0` which calls `expr1` etc... nesting more and more deeply. The precedence rule-of-thumb in ANTLR (and any LL-type parser) is: the deeper the nesting level, the higher the precedence (the more tightly the operator binds). Operators in the expression starting rule have the lowest precedence whereas operators in the last rule in the expression recursion have the highest precedence. This becomes obvious when a parse-tree for some input text is examined.

Once again, note that the operators for *sc* must be escaped as they are reserved regular expression operators as well.

2.4. Complete ANTLR description (w/o symbol table management)

The following code is the complete ANTLR description to recognize *sc* programs. Nothing is added to the symbol table and undefined variables/functions are not flagged.

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

#token "[\t\ ]+"    << zzskip(); >>                /* Ignore White */
#token "\n"         << zzline++; zzskip(); >>
#token STRING "\"(~[\0-\0x1f\"\\]|(\\~[\0-\0x1f]))*\"" <<;>>

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

p       :   ( func | "var" def ";" )*
            "@"
        ;

def     :   WORD
        ;

func    :   WORD "\(" { def } "\)"
            "\{"
                ( "var" def ";" )*
                ( statement )*
            "\}"
        ;

statement
        :   expr ";"
        |   "\{" ( statement )* "\}"
        |   "if" "\(" expr "\)" statement {"else" statement}
        |   "while" "\(" expr "\)" statement
        |   "return" expr ";"
        |   "print" expr ";"
        ;

expr    :   WORD "=" expr
        |   expr0
        ;

expr0   :   expr1 ( ("==" | "!=") expr1 )*
        ;

expr1   :   expr2 ( ("\+" | "\-") expr2 )*
        ;

expr2   :   expr3 ( ( "\*" | "/" ) expr3 )*
        ;

expr3   :   {"\-" } expr4
        ;

expr4   :   STRING
        |   WORD { "\(" { expr } "\)" }
        |   "\(" expr "\)"
        ;

#token WORD "[a-zA-Z]+"
```

2.5. Makefile

The following makefile can be used to make the above language description into an executable file. We assume that the ANTLR includes and standard attribute packages are located in a directory accessible to us as `tut1.g.`

```
#
# Makefile for 1.00 tutorial (no symbol table stuff)
# ANTLR creates parser.dlg, err.c, tut1.c, tokens.h
# DLG creates scan.c, mode.h
#
CFLAGS= -I../h
GRM=tut1
SRC=scan.c $(GRM).c err.c
OBJ=scan.o $(GRM).o err.o

tutorial: $(OBJ) $(SRC)
    cc -o $(GRM) $(OBJ)

# build a parser and lexical description from a language description
$(GRM).c parser.dlg : $(GRM).g
    antlr -k 2 $(GRM).g

# build the scanner from a lexical description
scan.c : parser.dlg
    dlg -C2 parser.dlg scan.c
```

Remember that `make` wants a tab, not spaces, in front of the action statements (e.g. `cc`, `antlr`, ...).

2.6. Testing

After successful completion of `make`, the executable `tut` will exist in the current directory. `tut` takes input from `stdin`. Therefore, one parses a file via

`tut <` *file*`.sc`

The prompt will return without a message if no syntax errors were discovered. However, if *file* contains lexical or syntactic errors, error messages will appear. We have given no instructions related to translation, so nothing happens if all is well.

2.7. Symbol table management

The grammar presented thus far can only recognize *sc* programs. No translation is possible because it does not deal with the semantics of the input program only its structure. To begin semantic interpretation, one must add new symbols to a symbol table. The symbols required to ‘‘understand’’ an *sc* program are

`VAR` Symbol is a local (denoted with the `#define` constant `LOCAL`), a parameter (`PARAMETER`) or global (`GLOBAL`) variable.

`FUNC` Symbol is a function whose level is always `GLOBAL`.

A symbol table record requires two fields: `token` which indicates either `VAR` or `FUNC` and `level` which is either `LOCAL`, `PARAMETER` or `GLOBAL`.

The PCCTS system comes with a simple symbol table manager. The source is documented well and its functions will be referenced without explanation here. To use the functions, our grammar must include a file called `sym.h` and define a symbol table structure. We shall put the `#include` in the `#header` directive.

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

The file `sym.c` contains the actual functions and must be linked into your executable. Our makefile will handle this automatically. The `sym.c` can be found in the `support/sym` subdirectory of the standard PCCTS installation; this should be copied into the tutorial directory.

The symbol table manager requires a number of fields within the symbol table entry. A template has been provided that the user can copy into their work directory and modify. The template in `sym.h` will be modified in our case to include the two fields mentioned above—`token` and `level`.

```
typedef struct symrec {
            char *symbol;
            struct symrec *next, *prev, **head, *scope;
            int token;  /* either FUNC or VAR */
            int level;  /* either LOCAL, GLOBAL, PARAMETER */
           int offset; /* offset from sp on the stack */
                       /* locals are - offset, param is 0 */
                       /* used only in tut4; reserved */
        } Sym, *SymPtr;
```

We add the following definitions to the front of our grammar.

```
<<
#define HashTableSize       999
#define StringTableSize     5000
#define GLOBAL              0
#define PARAMETER           1
#define LOCAL               2

static Sym *globals = NULL;        /* global scope for symbols */
>>
```

where `globals` is used to track all global symbols (functions and variables). Also, to print out symbol scopes, we define a function called `pScope(Sym *p)` that dumps a scope to `stdout`. It's implementation is unimportant and given with the full grammar description listed below. To initialize the symbol table, we add a function call to the symbol table manager library yielding a new `main()`.

```
main()
{
    zzs_init(HashTableSize, StringTableSize);
    ANTLR(p(), stdin);
}
```

When a variable, parameter or function is defined, we want to add that symbol to the symbol table. We shall treat parameters like variables grammatically and use field `level` to differentiate between them. When a `WORD` is found in rule `def`, we will add it to the symbol table using the scope and level passed into `def`. The situation is complicated slightly by the fact that a local variable may have the same name as a global variable. Scopes are linked lists that weave through the symbol table grouping all entries within the same scope. Our grammar for rule `def` becomes:

```
def[Sym **scope, int level] : <<Sym *var;>> (WORD | VAR) ;
```

where the init-action defines a local variable called `var`.

To handle the definition of previously unknown symbols, we add an action after the `WORD` reference.

```
( WORD
  <<zzs_scope($scope);         /* set current scope to scope passed in */
    var = zzs_newadd($1.text); /* create entry, add text of WORD to table */
    var->level = $level;       /* set the level to level passed in */
    var->token = VAR;          /* symbol is a variable */
  >>
| VAR
)
```

To deal with a symbol defined in another scope we add the following action to the `VAR` reference.

```
( WORD
  <<...>>
| VAR
  <<var = zzs_get($1.text);    /* get previous definition */
    if ( level != var->level ) /* make sure we have a diff scope */
    {
      zzs_scope($scope);       /* same here as above for unknown */
      var = zzs_newadd($1.text);
      var->level = $level;
      var->token = VAR;
    }
    else printf("redefined variable ignored: %s\n", $1.text);
  >>
)
```

Note that this implies that the lexical analyzer will modify the token number according to its definition in the symbol table (if any). This is generally not a good idea, but can be quite helpful when trying to remove nasty ambiguities in your grammar. Typically more than one token of lookahead makes it unnecessary to use ‘‘derived’’ tokens. We do so here to illustrate the interaction of lexical analyzer and parser.

Rule `p` must be modified to pass a scope and level to rule `def`. In addition, we will remove the global scope at the end of parsing and print out the symbols.

```
p       :   <<Sym *p;>>
            ( func | "var" def[&globals, GLOBAL] ";" )*
            <<p = zzs_rmscope(&globals);
              printf("Globals:\n");
              if ( p != NULL ) pScope(p);
            >>
            "@"
        ;
```

Rule `func` now must create a `FUNC` symbol table entry and define a `VAR` entry for its parameter if one exists. Rule `func` checks for duplicate `FUNC` entries by only matching unknown `WORD`'s. If a function had been previously defined, its token would be `FUNC`.

```
func    :   <<Sym *locals=NULL, *var, *p;>>
            WORD
             <<zzs_scope(&globals);
               var = zzs_newadd($1.text);
               var->level = GLOBAL;
               var->token = FUNC;
             >>
            "\(" { def[&locals, PARAMETER] } "\)"
            "\{"
                ( "var" def[&locals, LOCAL] ";" )*
                ( statement )*
            "\}"
            <<p = zzs_rmscope(&locals);
              printf("Locals for %s:\n", $1.text);
              if ( p != NULL ) pScope(p);
            >>
        ;
```

At the end of the function, we remove the local scope of variables (and parameter if it exists) and print the symbols out to `stdout`.

When a `WORD` is encountered on the input stream, we need to look it up in the symbol table to find out whether it is a variable (parameter) or a function. The token number needs to be changed accordingly before the parser sees it so that it will not try to match a `WORD`. Any `WORD` references in an expression that are not defined in the symbol table are undefined variables. We accomplish this token ‘‘derivation’’ strategy by attaching an action to the regular expression for `WORD`.

```
#token WORD "[a-zA-Z]+"
    <<{
        Sym *p = zzs_get(LATEXT(1));
        if ( p != NULL ) NLA = p->token;
    }>>
```

The macro `LATEXT(1)` is always set to the text matched on the input stream for the current token. `NLA` is the next token of look-ahead. We need to change this from `WORD` to whatever is found in the symbol table.

Rules for statements and expressions do not change when adding symbol table management because they simply apply a structure to grammar symbols and do not introduce new ones.

2.8. Complete ANTLR description (with symbol table management)

The following code is the complete ANTLR description to recognize *sc* programs. Functions, variables and parameters are added to the symbol table and are printed to `stdout` after function definitions and at the end of the *sc* program.

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

#token "[\t\ ]+"    << zzskip(); >>                /* Ignore White */
#token "\n"         << zzline++; zzskip(); >>
#token STRING "\"(~[\0-\0x1f\"\\]|(\\~[\0-\0x1f]))*\"" <<;>>

<<
#define HashTableSize       999
#define StringTableSize     5000
#define GLOBAL              0
#define PARAMETER           1
#define LOCAL               2

static Sym *globals = NULL; /* global scope for symbols */


main()
{
    zzs_init(HashTableSize, StringTableSize);
    ANTLR(p(), stdin);
}

pScope(p)
Sym *p;
{
    for (; p!=NULL; p=p->scope)
    {
        printf("\tlevel %d | %-12s | %-15s\n",
            p->level,
            zztokens[p->token],
            p->symbol);
    }
}
>>
```

```
p       :   <<Sym *p;>>
            ( func | "var" def[&globals, GLOBAL] ";" )*
            <<p = zzs_rmscope(&globals);
              printf("Globals:\n");
              pScope(p);
            >>
            "@"
        ;

def[Sym **scope, int level]
        :   <<Sym *var;>>
            (   WORD
                <<zzs_scope($scope);
                  var = zzs_newadd($1.text);
                  var->level = $level;
                  var->token = VAR;
                >>
            |   VAR
                <<var = zzs_get($1.text);
                  if ( $level != var->level )
                  {
                    zzs_scope($scope);
                    var = zzs_newadd($1.text);
                    var->level = $level;
                    var->token = VAR;
                  }
                  else printf("redefined variable ignored: %s\n", $1.text);
                >>
            )
        ;
```

```
func    :   <<Sym *locals=NULL, *var, *p;>>
            WORD
            <<zzs_scope(&globals);
              var = zzs_newadd($1.text);
              var->level = GLOBAL;
              var->token = FUNC;
            >>
            "\(" { def[&locals, PARAMETER] } "\)"
            "\{"
                ( "var" def[&locals, LOCAL] ";" )*
                ( statement )*
            "\}"
            <<p = zzs_rmscope(&locals);
              printf("Locals for %s:\n", $1.text);
              pScope(p);
            >>
        ;

statement
        :   expr ";"
        |   "\{" ( statement )* "\}"
        |   "if" "\(" expr "\)" statement
            {"else" statement}
        |   "while" "\(" expr "\)" statement
        |   "return" expr ";"
        |   "print" expr ";"
        ;


expr    :   VAR "=" expr
        |   expr0
        ;

expr0   :   expr1 ( (   "=="
                    |   "!="
                    )
                    expr1
                  )*
        ;

expr1   :   expr2 ( (   "\+"
                    |   "\-"
                    )
                    expr2
                  )*
        ;

expr2   :   expr3 ( (   "\*"
                    |   "/"
                    )
                    expr3
                  )*
        ;

expr3   :   {"\-" } expr4
        ;
```

```
expr4   :   STRING
        |   VAR
        |   (   FUNC
            |   WORD
            )
            "\(" { expr } "\)"
        |   "\(" expr "\)"
    ;

#token WORD "[a-zA-Z]+"
    <<{
        Sym *p = zzs_get(LATEXT(1));
        if ( p != NULL ) NLA = p->token;
    }>>
```

2.9. File `sym.h`

The following is a modification of the `sym.h` template provided with PCCTS. The fields of the symbol table entry structure have augmented for our purposes as outlined above.

```
/* define some hash function */
#ifndef HASH
#define HASH(p, h) while ( *p != '\0' ) h = (h<<1) + *p++;
#endif

typedef struct symrec {
            char * symbol;
            struct symrec *next, *prev, **head, *scope;
            unsigned hash;
            int token;  /* either FUNC or VAR */
            int level;  /* either LOCAL, GLOBAL, PARAMETER */
            int offset; /* offset from sp on the stack */
                        /* locals are - offset, param is 0 */
                        /* used only tut4; reserved */
        } Sym, *SymPtr;

void zzs_init();
void zzs_done();
void zzs_add();
Sym *zzs_get();
void zzs_del();
void zzs_keydel();
Sym **zzs_scope();
Sym *zzs_rmscope();
void zzs_stat();
Sym *zzs_new();
Sym *zzs_newadd();
char *zzs_strdup();
```

2.10. Makefile (for use with symbol table management)

The following makefile can be used to make the above language description into an executable file. We assume that the ANTLR includes and standard attribute packages are located in a directory accessible to us as `current working directory.  The grammar listed above must be in file tut2.g.`

```
#
# Makefile for 1.00 tutorial
# ANTLR creates parser.dlg, err.c, tut1.c
# DLG creates scan.c
#
CFLAGS= -I../h
GRM=tut2
SRC=scan.c $(GRM).c err.c sym.c
OBJ=scan.o $(GRM).o err.o sym.o

tutorial: $(OBJ) $(SRC)
    cc -o $(GRM) $(OBJ)

$(GRM).c parser.dlg : $(GRM).g
    antlr -k 2 $(GRM).g

scan.c : parser.dlg
    dlg -C2 parser.dlg scan.c
```

2.11. Sample input/output

The current state of the program accepts input like the following sample.

```
var i;
var j;

f(k)
{
    var local;
    var j;

    if ( "true" ) local = "zippo";
}

g()
{
    var note;

    note = "1";
    while ( note )
    {
        i = "456";
    }
}
```

The output of our executable, `tut`, would be

```
Locals for f:
        level 2 | VAR          | j
        level 2 | VAR          | local
        level 1 | VAR          | k
Locals for g:
        level 2 | VAR          | note
Globals:
        level 0 | FUNC         | g
        level 0 | FUNC         | f
        level 0 | VAR          | j
        level 0 | VAR          | i
```

Note that the parameter `k` is level 1 for `PARAMETER` and the local variable `local` is level 2 for `LOCAL`.

3. Translate *sc* to stack code

Generating code for a stack machine is simple and can be done by simply adding `printf()` actions to the grammar in the appropriate places.

We begin with a discussion of the stack machine and how to generate code for it. Next we augment our grammar with actions to dump stack code to `stdout`.

3.1. A simple stack machine for *sc*

Our stack machine consists of a single CPU, a finite stack of strings, a finite memory, a stack pointer (`sp`) and a frame pointer (`fp`). All data items used by stack programs are strings (currently set to a maximum length of 100). Our string stack grows downwards towards 0 from the maximum stack height.

To make implementation simple, our stack code will actually be a sequence of macro invocations in a C program. This way C will take care of control-flow and allocating space etc... The minimum stack code program defines a `_main` and includes `sc.h` which contains all of the stack code macros:

```
#include "sc.h"
_main()
{
    BEGIN;
    END;
}
```

The `BEGIN` and `END` macros are explained below.

3.1.1. Functions

Every *sc* program requires a `main` which will we translate to `_main` (a C function `main` will call `_main`). Other functions are translated verbatim. The parameter to your main program will be the first command line argument when you execute your *sc* program (after translation to C). If no command line argument is specified, a `""` is pushed.

Parameters are pushed on the string stack before a function is called, so no argument is needed to the resulting C function. Return values are implicitly strings but are also returned on the string stack — avoiding the need to define a return type of the C function. An ‘‘instruction’’ (macro) called `BEGIN` is executed before any other in a function. This saves the current frame pointer and then makes it point to the argument passed into the function. `END` is used to restore the frame pointer to its previous value and make the stack pointer point to the return value. After a function call, the top of stack is always the return value.

Arguments and local variables are at offsets to the frame pointer. The optional argument is at offset 0. The first local variable is at offset 1, the second at offset 2 and so on. Graphically,

```
|     .     |
|     .     |
|    arg    |   fp + 0
| 1st local |   fp - 1
| 2nd local |   fp - 2
|    ...    |
|     .     |
|     .     |
```

A dummy argument is pushed by the calling function if no argument is specified.

To translate a function,

```
f(arg)
{
}
```

we simply dump the following

```
f()
{
    BEGIN;
    END;
}
```

Note that the argument disappears because we pass arguments on the string stack not the C hardware stack.

3.1.2. Variables

Global variables of the form

```
var name;
```

are translated to

```
SCVAR name;
```

where SCVAR is a C type defined in `sc.h` that makes space for a *sc* variable.

Local variables are allocated on the string stack and are created at run time via the instruction `LOCAL`. Each execution of `LOCAL` allocates space for one more local variable on the stack. `LOCAL`'s must be executed after the `BEGIN` but before any other stack code instruction. The `END` macro resets the stack pointer so that these locals disappear after each function invocation. For example,

```
main()
{
    var i;
    var j;
}
```

is translated to:

```
#include "sc.h"
_main()
{
    BEGIN;
    LOCAL;
    LOCAL;
    END;
}
```

3.1.3. Expressions

Operator precedence is defined implicitly in top-down (LL) grammars. The deeper the level of recursion, the higher the precedence. To generate code for operators, one prints out the correct macro for that operator after the two operators have been seen (because we are generating code for a stack machine). If the operator is a unary operator like negate, we wait until the operand is seen and then print the operator. For instance, `expr1`,

```
expr1   :   expr2 ( ("\+"|"\-") expr2)*
        ;
```

would be translated as:

```
expr1   :   <<char *op;>>
            expr2 ( (   "\+" <<op="ADD";>>
                    |   "\-" <<op="SUB";>>
                    )
                    expr2
                    <<printf("\t%s;\n", op);>>
                  )*
        ;
```

where `ADD` and `SUB` are *sc* stack machine macros defined below.

The assignment operator groups right to left and must generate code that duplicates itself after each assignment so that the value of the expression is on the stack for any chained assignement. For example,

```
main()
{
    var a;
    var b;

    a = b = "test";
}
```

would be translated to:

```
#include "sc.h"
_main()             /* main() */
{
    BEGIN;
    LOCAL;          /* var a; */
    LOCAL;          /* var b; */
    SPUSH("test");  /* a = b = "test"; */
    DUP;
    LSTORE(2);      /* store into b */
    DUP;
    LSTORE(1);      /* store into a */
    POP;
    END;
}
```

Since an expression is a statement, it is possible that a value could be left on the stack. For example,

```
main()
{
    "expression";
}
```

We must pop this extraneous value off the stack:

```
#include "sc.h"
_main()
{
    BEGIN;
    SPUSH("expression");
    POP;
    END;
}
```

3.1.4. Instructions

All instructions take operands from the stack and return results on the stack. Some have no side effects on the stack but alter the C program counter.

`PUSH(`*v*`)` Push the value of variable *v* onto the stack.

`SPUSH(`*s*`)` Push the value of the string constant *s* onto the stack.

`LPUSH(`*n*`)` Push the value of the local variable or function argument at offset *n* onto the stack.

`STORE(`*v*`)` Pop the top of stack and store it into variable *v*.

`LSTORE(`*n*`)` Pop the top of stack and store it into local variable or function argument at offset *n*.

`POP` Pop the top of stack; stack is one string smaller. No side effects with data memory.

`LOCAL` Create space on the stack for a local variable. Must appear after `BEGIN` but before any other instruction in a function.

`BRF(`*a*`)` Branch, if top of stack is `"false"` or `"0"`, to ‘‘address’’ *a* (C label); top of stack is popped off.

`BRT(`*a*`)` Branch, if top of stack is `"true"` or `"1"`, to ‘‘address’’ *a* (C label); top of stack is popped off.

`BR(`*a*`)` Branch to ‘‘address’’ *a* (C label). Stack is not touched.

`CALL(`*f*`)` Call the function *f*. Returns with result on stack top.

`PRINT` Pop the top of stack and print the string to `stdout`.

`RETURN` Set the return value to the top of stack. Return from the function.

`EQ` Perform `stack[sp] = (stack[sp+1] == stack[sp])`.

`NEQ` Perform `stack[sp] = (stack[sp+1] != stack[sp])`.

`ADD` Perform `stack[sp] = (stack[sp+1] + stack[sp])`.

`SUB` Perform `stack[sp] = (stack[sp+1] - stack[sp])`.

`MUL` Perform `stack[sp] = (stack[sp+1] * stack[sp])`.

`DIV` Perform `stack[sp] = (stack[sp+1] / stack[sp])`.

`NEG` Perform `stack[sp] = -stack[sp]`.

`DUP` Perform `PUSH(stack[sp])`.

`BEGIN` Function preamble.

`END` Function cleanup.

Boolean operations yield string constants `"false"` for false and `true` for true. All other operations yield string constants representing numerical values (`"3"+"4"` is `"7"`).

3.2. Examples

The sample factorial program from above,

```
main(n)
{
    print fact(n);
    print "\n";
}

fact(n)
{
    if ( n == "0" ) return "1";
    return n * fact(n - "1");
}
```

would be translated to:

```
#include "sc.h"
_main()             /* main(n) */
{                   /* { */
    BEGIN;
    LPUSH(0);       /* print fact(n); */
    CALL(fact);
    PRINT;
    SPUSH("\n");    /* print "\n"; */
    PRINT;
    END;            /* } */
}


fact()              /* fact(n) */
{
    BEGIN;
    LPUSH(0);       /* if ( n == "0" ) */
    SPUSH("0");
    EQ;
    BRF(iflabel0);
    SPUSH("1");     /* return "1"; */
    RETURN;
iflabel0: ;
    LPUSH(0);       /* return n * fact(n - "1"); */
    LPUSH(0);
    SPUSH("1");
    SUB;
    CALL(fact);
    MUL;
    RETURN;
    END;            /* } */
}
```

3.3. Augmenting grammar to dump stack code

In order to generate code, we must track the offset of local variables. To do so, we add a field, `offset`, to our symbol table record:

```
typedef struct symrec {
    char * symbol;
    struct symrec *next, *prev, **head, *scope;
    unsigned hash;
    int token;  /* either FUNC or VAR */
    int level;  /* either LOCAL, GLOBAL, PARAMETER */
    int offset; /* offset from sp on the stack */
                /* locals are negative offsets, param is 0 */
    } Sym, *SymPtr;
```

We begin modifying our grammar by making a global variable that tracks the offset of local variables and defining a variable to track label numbers:

```
static int current_local_var_offset, LabelNum=0;
```

Because the translation of all programs must include the *sc* stack machine definitions, we add a `printf` to rule `p`:

```
p       :   <<Sym *p; printf("#include \"sc.h\"\n"); >>
            ( func | "var" def[&globals, GLOBAL] ";" )*
            <<  p = zzs_rmscope(&globals); >>
            "@"
        ;
```

In order to generate the variable definition macros and update the symbol table, we modify rule `def` as follows:

```
def[Sym **scope, int level]
        :   <<Sym *var;>>
            (   WORD
                <<zzs_scope($scope);
                  var = zzs_newadd($1.text);
                  var->level = $level;
                  var->token = VAR;
                  var->offset = current_local_var_offset++;
                  switch(var->level) {
                        case GLOBAL: printf("SCVAR %s;\n", $1.text); break;
                        case  LOCAL : printf("\tLOCAL;\n"); break;
                  }
                >>
            |   VAR
                <<var = zzs_get($1.text);
                  if ( $level != var->level )
                  {
                    zzs_scope($scope);
                    var = zzs_newadd($1.text);
                    var->level = $level;
                    var->token = VAR;
                    var->offset = current_local_var_offset++;

                    switch(var-> level) {
                        case GLOBAL: printf("\tSCVAR %s;\n",$1.text);break;
                        case  LOCAL : printf("\tLOCAL;\n"); break;
                    }
                  }
                  else printf("redefined variable ignored: %s\n",$1.text);
                >>
            )
        ;
```

The function definition rule must now dump a function template to `stdout` and generate the `BEGIN` and `END` macros. `func` must also update `current_local_var_offset`. Note that the code to dump symbols is gone.

```
func :  <<Sym *locals=NULL, *var, *p; current_local_var_offset =0;>>
            WORD
            <<zzs_scope(&globals);
              var = zzs_newadd($1.text);
              var->level = GLOBAL;
              var->token = FUNC;

              if (strcmp("main",$1.text)) { printf("%s()\n",$1.text); }
              else printf("_main()\n");
            >>
            "\(" (  def[&locals, PARAMETER]
                 |  <<current_local_var_offset = 1;>>
                 )
            "\)"
            "\{" << printf("{\n\tBEGIN;\n"); >>
                ( "var" def[&locals, LOCAL] ";" )*
                ( statement )*
            "\}" << printf("\tEND;\n}\n"); >>
            <<  p = zzs_rmscope(&locals); >>
        ;
```

Statements are easy to handle since `expr` generates most of the code.

```
statement : <<int n;>>
            expr ";"            <<printf("\tPOP;\n");>>
        |   "\{" ( statement )* "\}"
        |   "if" "\(" expr "\)" << n = LabelNum++;
                                   printf("\tBRF(iflabel%d);\n",n); >>
                statement       << printf("iflabel%d: ;\n",n); >>
            {"else" statement}
        |
            "while"             << n = LabelNum++;
                                   printf("wbegin%d: ;\n", n); >>
            "\(" expr "\)"      << printf("\tBRF(wend%d);\n",n); >>
            statement           << printf("\tBR(wbegin%d);\n", n); >>
                                << printf("wend%d: ;\n",n); >>
                                << n++; >>
        |   "return" expr ";"   << printf("\tRETURN;\n"); >>
        |   "print" expr ";"    << printf("\tPRINT;\n"); >>
        ;
```

3.4. Full translator

The following grammar accepts programs in *sc* and translates them into stack code.

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

#token "[\t\ ]+"  << zzskip(); >>                /* Ignore White */
#token "\n"       << zzline++; zzskip(); >>
#token STRING     "\"(~[\0-\0x1f\"\\]|(\\~[\0-\0x1f]))*\"" <<;>>

<<
#define HashTableSize       999
#define StringTableSize     5000
#define GLOBAL              0
#define PARAMETER           1
#define LOCAL               2

static Sym *globals = NULL; /* global scope for symbols */
static int current_local_var_offset, LabelNum=0;


main()
{
    zzs_init(HashTableSize, StringTableSize);
    ANTLR(p(), stdin);
}

pScope(p)
Sym *p;
{
    for (; p!=NULL; p=p->scope)
    {
        printf("\tlevel %d | %-12s | %-15s\n",
            p->level,
            zztokens[p->token],
            p->symbol);
    }
}
>>
```

```
p       :   <<Sym *p; printf("#include \"sc.h\"\n"); >>
            ( func | "var" def[&globals, GLOBAL] ";" )*
            <<  p = zzs_rmscope(&globals); >>
            "@"
        ;

def[Sym **scope, int level]
        :   <<Sym *var;>>
            (   WORD
                <<zzs_scope($scope);
                  var = zzs_newadd($1.text);
                  var->level = $level;
                  var->token = VAR;
                  var->offset = current_local_var_offset++;
                  switch(var->level) {
                        case  GLOBAL: printf("SCVAR %s;\n", $1.text); break;
                        case  LOCAL : printf("\tLOCAL;\n"); break;
                  }
                >>
            |   VAR
                <<var = zzs_get($1.text);
                  if ( $level != var->level )
                  {
                    zzs_scope($scope);
                    var = zzs_newadd($1.text);
                    var->level = $level;
                    var->token = VAR;
                    var->offset = current_local_var_offset++;

                    switch(var-> level) {
                        case  GLOBAL: printf("\tSCVAR %s;\n", $1.text); break;
                        case  LOCAL : printf("\tLOCAL;\n"); break;
                    }
                  }
                  else printf("redefined variable ignored: %s\n", $1.text);
                >>
            )
        ;
```

```
func    :   <<Sym *locals=NULL, *var, *p; current_local_var_offset = 0;>>
            WORD
            <<zzs_scope(&globals);
              var = zzs_newadd($1.text);
              var->level = GLOBAL;
              var->token = FUNC;

              if (strcmp("main",$1.text)) { printf("%s()\n",$1.text); }
              else printf("_main()\n");
            >>
            "\(" (  def[&locals, PARAMETER]
                 |  <<current_local_var_offset = 1;>>
                 )
            "\)"
            "\{" << printf("{\n\tBEGIN;\n"); >>
                ( "var" def[&locals, LOCAL] ";" )*
                ( statement )*
            "\}" << printf("\tEND;\n}\n"); >>
            <<  p = zzs_rmscope(&locals); >>
        ;

statement : <<int n;>>
            expr ";"            <<printf("\tPOP;\n");>>
        |   "\{" ( statement )* "\}"
        |   "if" "\(" expr "\)" << n = LabelNum++;
                                   printf("\tBRF(iflabel%d);\n",n); >>
                statement       << printf("iflabel%d: ;\n",n); >>
            {"else" statement}
        |
            "while"             << n = LabelNum++;
                                   printf("wbegin%d: ;\n", n); >>
            "\(" expr "\)"      << printf("\tBRF(wend%d);\n",n); >>
            statement           << printf("\tBR(wbegin%d);\n", n); >>
                                << printf("wend%d: ;\n",n); >>
                                << n++; >>
        |   "return" expr ";"   << printf("\tRETURN;\n"); >>
        |   "print" expr ";"    << printf("\tPRINT;\n"); >>
        ;
```

```
expr    :   <<Sym *s;>>
            VAR "=" expr        << printf("\tDUP;\n"); >>
            <<s = zzs_get($1.text);
              if ( s->level != GLOBAL )
                printf("\tLSTORE(%d);\n", s->offset);
              else printf("\tSTORE(%s);\n", s->symbol);
            >>
        |   expr0
        ;

expr0   :   <<char *op;>>
            expr1 ( (   "==" <<op="EQ";>>
                    |   "!=" <<op="NEQ";>>
                    )
                    expr1
                    <<printf("\t%s;\n", op);>>
                  )*
        ;

expr1   :   <<char *op;>>
            expr2 ( (   "\+" <<op="ADD";>>
                    |   "\-" <<op="SUB";>>
                    )
                    expr2
                    <<printf("\t%s;\n", op);>>
                  )*
        ;

expr2   :   <<char *op;>>
            expr3 ( (   "\*" <<op="MUL";>>
                    |   "/"  <<op="DIV";>>
                    )
                    expr3
                    <<printf("\t%s;\n", op);>>
                  )*
        ;
```

```
expr3   :   <<char *op=NULL;>>
            {"\-" <<op="NEG";>>} expr4
            <<if ( op!=NULL ) printf("\t%s;\n", op);>>
        ;

expr4   :   <<Sym *s; int arg;>>
            STRING   << printf("\tSPUSH(%s);\n", $1.text); >>
        |   VAR     << s = zzs_get($1.text);
                       if ( s->level != GLOBAL )
                            printf("\tLPUSH(%d);\n", s->offset);
                       else printf("\tPUSH(%s);\n", s->symbol);
                    >>
        |   (   FUNC    <<$0=$1;>>
            |   WORD    <<$0=$1;>>
            )
            "\(" { <<arg=0;>> expr <<arg=1;>> } "\)"
            <<if ( !arg ) printf("\tSPUSH(\"\");\n");
              printf("\tCALL(%s);\n", $1.text);>>
        |   "\(" expr "\)"
    ;

#token WORD "[a-zA-Z]+"
    <<{
        Sym *p = zzs_get(zzlextext);
        if ( p != NULL ) NLA = p->token;
    }>>
```

3.5. Makefile for Full Translator

The makefile is no different from the one used for the symbol table management version of the tutorial except that we need to change the `GRM` make variable as follows:

```
GRM=tut3
```

which implies that `tut3.g` is the file containing the third revision of our *sc* translator.

3.6. Use of translator

Once the translator has been made using the makefile, it is ready to translate *sc* programs. To translate and execute the factorial example from above, we execute the following:

```
tut3 < fact.c > temp.c
```

where `fact.c` is the file containing the factorial code. `temp.c` will contain the translated program. It is valid C code because it is nothing more than a bunch of macro invocations (the macros are defined in `sc.h`). We can compile `temp.c` like any other C program:

```
cc temp.c
```

To execute the program, we type:

```
a.out n
```

where *n* is the number you want the factorial of. Typing:

```
a.out 8
```

yields:

```
40320.000000
```

3.7. Stack code macros — `sc.h`

The following file is included by any C program using the stack code instructions outlined above.

```
/* sc.h -- string C stack machine macros
 *
 * For use with PCCTS advanced tutorial version 1.0x
 */
#include <stdio.h>
#include <ctype.h>
#include <math.h>

/*
 * The function invocation stack looks like:
 *
 *  |    .       |
 *  |    .       |
 *  |    arg     |    fp + 0       arg is "" if none specified
 *  | 1st local  |    fp - 1
 *  | 2nd local  |    fp - 2
 *  |    ...     |
 *  |    .       |
 *  |    .       |
 */


#define STR_SIZE     100
#define STK_SIZE     200
/* define stack */
typedef struct { char text[STR_SIZE]; } SCVAR;
static SCVAR stack[STK_SIZE];
static int sp = STK_SIZE, fp;

/* number begins with number or ’.’ followed by number.  All numbers
 * are converted to floats before comparison.
 */
#define SCnumber(a)  (isdigit(a[0]) || (a[0]==’.’ && isdigit(a[1])))
```

```
#define TOS          stack[sp].text
#define NTOS     stack[sp+1].text
#define TOSTRUE      ((strcmp(TOS, "true")==0)||(strcmp(TOS, "1")==0)\
                     ||(SCnumber(TOS)&&atof(TOS)==1.0) )
#define TOSFALSE ((strcmp(TOS, "false")==0)||(strcmp(TOS, "0")==0)\
                     ||(SCnumber(TOS)&&atof(TOS)==0.0) )

#define PUSH(a)      {if ( sp==0 ) {fprintf(stderr, "stk ovf!\n"); exit(-1);} \
                     strcpy(stack[--sp].text, (a).text);}

#define SPUSH(a) {if ( sp==0 ) {fprintf(stderr, "stk ovf!\n"); exit(-1);} \
                     strcpy(stack[--sp].text, a);}

#define LPUSH(a) {if ( sp==0 ) {fprintf(stderr, "stk ovf!\n"); exit(-1);} \
                     stack[--sp] = stack[fp-a];}


#define CALL(f)      {f();}

#define POP          stack[sp++]

#define LOCAL        {if ( sp==0 ) {fprintf(stderr, "stk ovf!\n"); exit(-1);} \
                     stack[--sp].text[0] = '\0';}


#define BRF(lab) if ( TOSFALSE ) {POP; goto lab;} else POP;
#define BRT(lab) if ( TOSTRUE ) {POP; goto lab;} else POP;
#define BR(lab)      goto lab

#define PRINT        printf("%s", POP.text)

#define RETURN       {strcpy(stack[fp].text, TOS); END; return;}

#define STORE(v) {v = POP;}
#define LSTORE(off)  {stack[fp-off] = POP;}
```

```
/* operators */

#define EQ          {char *a,*b; float c,d;                  \
                    a = POP.text; b = POP.text;              \
                    if ( SCnumber(a) && SCnumber(b) ) {      \
                        c=atof(a); d=atof(b);                \
                        if ( c == d ) {SPUSH("true");}       \
                        else SPUSH("false");                 \
                    }                                        \
                    else if ( strcmp(a, b)==0 ) {SPUSH("true");}\
                         else SPUSH("false");}
#define NEQ         {SCVAR a,b; float c,d;                   \
                    a = POP.text; b = POP.text;              \
                    if ( SCnumber(a) && SCnumber(b) ) {      \
                        c=atof(a); d=atof(b);                \
                        if ( c != d ) {SPUSH("true");}       \
                        else SPUSH("false");                 \
                    }                                        \
                    else if ( strcmp(a, b)!=0 ) {SPUSH("true");}\
                         else SPUSH("false");}
#define ADD         {SCVAR c; float a,b;                     \
                    if ( !SCnumber(NTOS) || !SCnumber(TOS) ) { \
                        strncat(NTOS, TOS, STR_SIZE - strlen(NTOS));\
                        sp++;                                \
                    } else {                                 \
                        a=atof(POP.text); b=atof(POP.text);  \
                        sprintf(c.text, "%f", a+b); PUSH(c); \
                    }}
#define SUB         {SCVAR c; float a,b; a=atof(POP.text); b=atof(POP.text); \
                    sprintf(c.text, "%f", b-a); PUSH(c);}
#define MUL         {SCVAR c; float a,b; a=atof(POP.text); b=atof(POP.text); \
                    sprintf(c.text, "%f", a*b); PUSH(c);}
#define DIV         {SCVAR c; float a,b; a=atof(POP.text); b=atof(POP.text); \
                    sprintf(c.text, "%f", b/a); PUSH(c);}
#define NEG         {SCVAR c; float a; a=atof(POP.text); \
                    sprintf(c.text, "%f", -a); PUSH(c);}
#define DUP         {if ( sp==0 ) {fprintf(stderr, "stk ovf!\n"); exit(-1);} \
                    stack[sp-1] = stack[sp]; --sp;}


#define BEGIN       int save_fp = fp; fp = sp
#define END         sp = fp; fp = save_fp;

main(argc, argv)
int argc;
char *argv[];
{
    if ( argc == 2 ) {SPUSH(argv[1]);}
    else SPUSH("");
    CALL(_main);
    POP;
}
```

4. Intermediate form construction

This section describes how trees can be used as an intermediate form of the *sc* source program and how our `tut2.g` symbol table management example can be modified to construct these trees. Example tree constructions are given as well as the complete source.

Note that this section and the code generation section essentially duplicate the translation achieved in the previous section on stack code. We arrive at the solution from a different angle, however. This method is much more general and would be used for ‘‘real’’ grammars.

4.1. Tree Construction

To construct an intermediate form (IF) representation of an *sc* program, we will construct trees using the abstract syntax tree (AST) mechanism provided with PCCTS. PCCTS supports an automatic and an explicit tree construction mechanism; we will use both. We will augment the grammar that has the symbol table management actions — `tut2.g`.

In order to use the AST mechanism within PCCTS, we must do the following:

- Define the AST fields, `AST_FIELDS`, needed by the user.
- Define the default AST node constructor, `zzcr_ast()`, that converts from a lexeme to an AST node.
- Define an explicit AST node constructor — `zzmk_ast()` (because we use the explicit tree constructor mechanism also).

Our tree nodes will contain a token number to identify the node. This token number can also be a value for which there is no corresponding lexical item (e.g. a function call may have a node called `DefineVar`). If the token indicates that the node represents a variable or function, we must have information as to its level, offset from the frame pointer (if a local or parameter) and the string representing the name of the item. This information can all be held via:

```
#define AST_FIELDS int token, level, offset; char str[D_TextSize];
```

which will be added to the `#header` PCCTS pseudo-op action. Note that `D_TextSize` is defined in `charbuf`.h.

To tell PCCTS how it should build AST nodes for use with the automatic tree construction mechanism, we define the following macro in the `#header` action:

```
#define zzcr_ast(ast,attr,tok,text) create_ast(ast,attr,tok,text)
```

which merely calls a function we define as follows:

```
create_ast(ast,attr,tok,text)
AST *ast;
Attrib *attr;
int tok;
char *text;
{
    Sym *s;

    ast->token = tok;
    if ( tok == VAR )
    {
        s = zzs_get(text);
        ast->level = s->level;
        ast->offset = s->offset;
    }
    if ( tok == STRING || tok == VAR || tok == FUNC ) strcpy(ast->str, text);
}
```

Because of the symbol table lookup etc... we make the node constructor a function rather than have the code replicated at each invocation of the `zzcr_ast` macro.

When we explicitly construct trees, the automatic node constructor is generally disabled and we must explicitly call a function to create an AST node:

```
AST *
zzmk_ast(node, token, str)
AST *node;
int token;
char *str;
{
    Sym *s;

    node->token = token;
    if ( token == VAR )
    {
        s = zzs_get(str);
        node->level = s->level;
        node->offset = s->offset;
    }
    if ( tok == STRING || tok == VAR || tok == FUNC )
         strcpy(node->str, str);
    return node;
}
```

This function will be invoked when we have a grammar action with a reference of the form `#[`*token, string*`]`.

In order to print out the trees that we have defined, let us define a simple function to do a preorder, lisp-like walk of our trees:

```
lisp(tree)
AST *tree;
{
    while ( tree!= NULL )
    {
        if ( tree->down != NULL ) printf(" (");
        if ( tree->token == STRING ‖
             tree->token == VAR ‖
             tree->token == FUNC ) printf(" %s", tree->str);
        else printf(" %s", zztokens[tree->token]);
        lisp(tree->down);
        if ( tree->down != NULL ) printf(" )");
        tree = tree->right;
    }
}
```

PCCTS automatically assumes that all rules will produce trees and that all terminals within rules will result in nodes added to the current tree. In the case of rules `p`, `expr4`, `statement` and `func`, this is not true. We will explicitly handle the trees within these rules. Therefore, we append a `!` after the definition of these rules. We need to decide how each *sc* construct will be translated to a subtree and when we will pass these trees off to the code generator. We will collect all subtrees within a function and then, just before we remove the symbols for that function, we will pass this tree off to `lisp` to be printed; we will pass it off to the code generator when we define it in the next section. After it has been printed, the function trees will be destroyed. For global variable definitions, we will pass each one individually off to `lisp` and then destroy the tree.

Before modifying the rules of the grammar, we must define labels for a number of regular expressions so that the token value of that expression can be referenced from within an action. We also define labels for tokens which do not exist physically in a program, but are needed for code generation.

```
/* Not actual terminals, just node identifiers */
#token DefineFunc
#token SLIST


/* Define tokens for code generation */
#token DefineVar    "var"
#token Mul          "
#token Div          "/"
#token Add          "+"
#token Sub          "-"
#token Equal        "=="
#token NotEqual     "!="
#token If           "if"
#token While        "while"
#token Return       "return"
#token Print        "print"
#token Assign       "="
```

All PCCTS grammars that use the AST mechanism need to pass the address of a `NULL` pointer to the starting rule.

```
main()
{
    AST *root=NULL;

    zzs_init(HashTableSize, StringTableSize);
    ANTLR(p(&root), stdin);
}
```

Rule `p` is modified as follows:

```
p!       :   <<Sym *p; AST *v;>>
             (   func
             |   "var" def[&globals, GLOBAL] ";"
                 <<v = #(#[DefineVar], #2);
                   lisp(v); printf("\n"); zzfree_ast(v);
                 >>
             )*
             <<p = zzs_rmscope(&globals);>>
             "@"
         ;
```

The `#[DefineVar]` reference calls our `zzmk_ast` macro to create an AST node whose token is set to `DefineVar`. The reference to `#(#[DefineVar], #2)` is a tree constructor which makes the `DefineVar` node the parent of the tree returned by rule `def` — `#2`. In general, the tree constructor has the form:

`#(` *root*`,` *sibling1*`,` *sibling2*`, ...,` *siblingN* `)`

where any `NULL` pointer terminates the list except for the root pointer. `#2` refers to the second AST node or tree within an alternative — `def` in our case. Rule `def` itself does not change because PCCTS automatically creates a node for the `WORD` or `VAR` and passes it back to the invoking rule.

Trees for functions are constructed in the following form:

```
DefineFunc
    |
    v
   FUNC -> DefineVar -> DefineVar -> SLIST
                |            |          |
                v            |          v
          optional_arg       |        stat1 -> ... > statn
                             v
                           local1 -> ... -> localn
```

where `DefineFunc` and `DefineVar` are dummy tokens used only in trees (as opposed to the grammar). Rule `func` is modified to build these trees:

```
func!  :  <<Sym *locals=NULL, *var, *p; AST *f,*parm=NULL,*v=NULL,*s=NULL;
            current_local_var_offset = 0;>>
          WORD
          <<zzs_scope(&globals);
            var = zzs_newadd($1.text);
            var->level = GLOBAL;
            var->token = FUNC;
          >>
          "\(" (  def[&locals, PARAMETER] <<parm=#1;>>
              |  <<current_local_var_offset = 1;>>
              )
          "\)"
          "\{"
              ( "var" def[&locals, LOCAL]
                <<if ( v==NULL ) v = #2; else v = #(NULL,v,#2);>>
                ";"
              )*
              ( statement
                <<if ( s==NULL ) s = #1; else s = #(NULL,s,#1);>>
              )*
          "\}"
          <<s = #(#[SLIST], s);
            v = #(#[DefineVar], v);
            parm = #(#[DefineVar], parm);
            f = #(#[DefineFunc], #[FUNC,$1.text], parm, v, s);
            lisp(f); printf("\en"); zzfree_ast(f);
            p = zzs_rmscope(&locals);>>
        ;
```

Variable `parm` is set to the AST node created by `def` if a parameter is found else it is `NULL`. Variable `v` is used to maintain a list of local variables. The tree constructor

```
v = #(NULL,v,#2);
```

says to make a new tree with no root and with siblings `v` and `#2` — which effectively links `#2` to the end of the current list of variables. Variable `s` behaves in a similar fashion. After the list of variables and statements have been accumulated, we add a dummy node (`DefineVar`, `SLIST`) as a root to each list so that we get the structure given above. To examine the trees we have constructed, we make a call to `lisp` and then destroy the tree.

Even though it is not necessary to do so, we will build trees for `statement` explicitly (in general, the automatic mechanism is best for expressions with simple operator/operand relationships).

```
statement!: <<AST *s=NULL, *el=NULL;>>
            expr ";"                         <<#0 = #1;>>
        |   "\{"
                ( statement
                  <<if ( s==NULL ) s = #1; else s = #(NULL,s,#1);>>
                )*
            "\}" <<#0 = #(#[SLIST], s);>>
        |   "if" "\(" expr "\)" statement
            {"else" statement <<el=#2;>>}     <<#0 = #(#[If], #3, #5,el);>>
        | "while" "\(" expr "\)" statement    <<#0 = #(#[While], #3,#5);>>
        | "return" expr ";"                   <<#0 = #(#[Return], #2);>>
        | "print" expr ";"                    <<#0 = #(#[Print], #2);>>
        ;
```

Many of the tokens in `statement` are not included in the tree because they are a notational convenience for humans or are used for grouping purposes. As before, we track lists of statements using the tree constructor: `#(NULL,s,#1)`.

The last unusual rule is `expr`:

```
expr4!  :   <<AST *f=NULL, *arg=NULL;>>
            STRING          <<#0 = #[STRING, $1.text];>>
        |   VAR             <<#0 = #[STRING, $1.text];>>
        |   (   FUNC        <<f  = #[FUNC, $1.text];>>
            |   WORD        <<f  = #[FUNC, $1.text];>>
            )
            "\(" { expr <<arg=#1;>> } "\)"
            <<#0 = #(f, arg);>>
        |   "\(" expr "\)"  <<#0 = #2;>>
        ;
```

Assigning a tree to `#0` makes that tree the return tree. However, it should really only be used in rules that do not use the automatic tree construction mechanism; i.e. there is a `! after` the rule definition. For a function call, we make a tree with a `FUNC` node at the root and the optional argument as the child. The third alternative merely returns what is computed by `expr`; the parenthesis are mechanism for syntactic precedence and add nothing to the tree. The tree structure itself embodies the precedence.

The only other required modifications to the grammar are to indicate which tokens are operators (subtree roots) and which are operands in the expression rules. To indicate that a token is to be made a subtree root, we append a `^`. All other tokens are considered operands (children). Appending a token with `!` indicates that it is not to be included in the tree. For example, to make trees like:

```
operator
    |
    v
  opnd1 -> opnd2
```

we modify the operators in `expr` through `expr3` in a fashion similar to:

```
expr1   :   expr2 ( (    "\+"^
                    |   "\-"^
                    )
                    expr2
                  )*
        ;
```

and

```
expr3   :   { "\-"^ } expr4
        ;
```

4.2. Full Grammar to Construct Trees

The following grammar constructs AST's as an intermediate form (IF) of an *sc* program and prints out the IF in lisp-like preorder.

```
#header <<#include "sym.h"
          #include "charbuf.h"
          #define AST_FIELDS int token, level, offset; char str[D_TextSize];
          #define zzcr_ast(ast,attr,tok,text) create_ast(ast,attr,tok,text)
          #define DONT_CARE         0
          #define SIDE_EFFECTS      1
          #define VALUE             2
        >>

#token "[\t\ ]+"    << zzskip(); >>                /* Ignore White */
#token "\n"         << zzline++; zzskip(); >>
#token STRING "\"(~[\0-\0x1f\"\\]|(\\~[\0-\0x1f]))*\"" <<;>>

/* Not actual terminals, just node identifiers */
#token DefineFunc
#token SLIST
```

```
/* Define tokens for code generation */
#token DefineVar    "var"
#token Mul          "\*"
#token Div          "/"
#token Add          "\+"
#token Sub          "\-"
#token Equal        "=="
#token NotEqual     "!="
#token If           "if"
#token While        "while"
#token Return       "return"
#token Print        "print"
#token Assign       "="

<<;>>

<<
#define HashTableSize       999
#define StringTableSize     5000
#define GLOBAL              0
#define PARAMETER           1
#define LOCAL               2


static Sym *globals = NULL; /* global scope for symbols */
static int current_local_var_offset=0;

create_ast(ast,attr,tok,text)
AST *ast;
Attrib *attr;
int tok;
char *text;
{
    Sym *s;

    ast->token = tok;
    if ( tok == VAR )
    {
        s = zzs_get(text);
        ast->level = s->level;
        ast->offset = s->offset;
    }
    if ( tok == STRING ‖ tok == VAR ‖ tok == FUNC ) strcpy(ast->str, text);
}
```

```
AST *
zzmk_ast(node, token, str)
AST *node;
int token;
char *str;
{
    Sym *s;

    node->token = token;
    if ( token == VAR )
    {
        s = zzs_get(str);
        node->level = s->level;
        node->offset = s->offset;
    }
    if ( token == STRING ‖ token == VAR ‖ token == FUNC )
        strcpy(node->str, str);
    return node;
}


lisp(tree)
AST *tree;
{
    while ( tree!= NULL )
    {
        if ( tree->down != NULL ) printf(" (");
        if ( tree->token == STRING ‖
             tree->token == VAR ‖
             tree->token == FUNC ) printf(" %s", tree->str);
        else printf(" %s", zztokens[tree->token]);
        lisp(tree->down);
        if ( tree->down != NULL ) printf(" )");
        tree = tree->right;
    }
}

main()
{
    AST *root=NULL;

    zzs_init(HashTableSize, StringTableSize);
    ANTLR(p(&root), stdin);
}
```

```
pScope(p)
Sym *p;
{
    for (; p!=NULL; p=p->scope)
    {
        printf("\tlevel %d | %-12s | %-15s\n",
            p->level,
            zztokens[p->token],
            p->symbol);
    }
}
>>

p!      :   <<Sym *p; AST *v;>>
            (   func
            |   "var" def[&globals, GLOBAL] ";"
                <<v = #(#[DefineVar], #2);
                  gen(v,DONT_CARE); printf("\n"); zzfree_ast(v);
                >>
            )*
            <<p = zzs_rmscope(&globals);>>
            "@"
        ;
```

```
def[Sym **scope, int level]
        :   <<Sym *var;>>
            (   WORD
                <<zzs_scope($scope);
                  var = zzs_newadd($1.text);
                  var->level = $level;
                  var->token = VAR;
                  var->offset = current_local_var_offset++;
                >>
            |   VAR
                <<var = zzs_get($1.text);
                  if ( $level != var->level )
                  {
                    zzs_scope($scope);
                    var = zzs_newadd($1.text);
                    var->level = $level;
                    var->token = VAR;
                    var->offset = current_local_var_offset++;
                  }
                  else printf("redefined variable ignored: %s\n", $1.text);
                >>
            )
        ;

func!   :   <<Sym *locals=NULL, *var, *p; AST *f,*parm=NULL,*v=NULL,*s=NULL;
              current_local_var_offset = 0;>>
            WORD
            <<zzs_scope(&globals);
              var = zzs_newadd($1.text);
              var->level = GLOBAL;
              var->token = FUNC;
            >>
            "\(" (  def[&locals, PARAMETER] <<parm=#1;>>
                 |  <<current_local_var_offset = 1;>>
                 )
            "\)"
            "\{"
                ( "var" def[&locals, LOCAL]
                  <<if ( v==NULL ) v = #2; else v = #(NULL,v,#2);>>
                  ";"
                )*
                ( statement
                  <<if ( s==NULL ) s = #1; else s = #(NULL,s,#1);>>
                )*
            "\}"
            <<s = #(#[SLIST], s);
              v = #(#[DefineVar], v);
              parm = #(#[DefineVar], parm);
              f = #(#[DefineFunc], #[FUNC,$1.text], parm, v, s);
              gen(f,DONT_CARE); printf("\n"); zzfree_ast(f);
              p = zzs_rmscope(&locals);>>
        ;
```

```
statement!: <<AST *s=NULL, *el=NULL;>>
            expr ";"                            <<#0 = #1;>>
        |   "\{"
                ( statement
                  <<if ( s==NULL ) s = #1; else s = #(NULL,s,#1);>>
                )*
            "\}"                                <<#0 = #(#[SLIST], s);>>
        |   "if" "\(" expr "\)" statement
            {"else" statement <<el=#2;>>}       <<#0 = #(#[If], #3, #5, el);>>
        |   "while" "\(" expr "\)" statement   <<#0 = #(#[While], #3, #5);>>
        |   "return" expr ";"                  <<#0 = #(#[Return], #2);>>
        |   "print" expr ";"                   <<#0 = #(#[Print], #2);>>
        ;

expr    :   VAR "="^ expr
        |   expr0
        ;

expr0   :   expr1 ( (   "=="^
                    |   "!="^
                    )
                    expr1
                  )*
        ;

expr1   :   expr2 ( (   "\+"^
                    |   "\-"^
                    )
                    expr2
                  )*
        ;

expr2   :   expr3 ( (   "\*"^
                    |   "/"^
                    )
                    expr3
                  )*
        ;
```

```
expr3   :   { "\-"^ } expr4
        ;

expr4!  :   <<AST *f=NULL, *arg=NULL;>>
            STRING          <<#0 = #[STRING, $1.text];>>
        |   VAR             <<#0 = #[VAR, $1.text];>>
        |   (   FUNC        <<f  = #[FUNC, $1.text];>>
            |   WORD        <<f  = #[FUNC, $1.text];>>
            )
            "\(" { expr <<arg=#1;>> } "\)"
            <<#0 = #(f, arg);>>
        |   "\(" expr "\)"  <<#0 = #2;>>
        ;

#token WORD "[a-zA-Z]+"
    <<{
        Sym *p = zzs_get(LATEXT(1));
        if ( p != NULL ) NLA = p->token;
    }>>
```

4.3. Example translations

To illustrate how functions get translated, consider the following *sc* program:

```
var a;

main(n)
{
    var b;
}
```

Our translator would create trees represented by:

```
 ( DefineVar WORD )
 ( DefineFunc FUNC ( DefineVar WORD ) ( DefineVar WORD ) SLIST )
```

where `WORD` represents the variables found on the input stream. `SLIST` has no child because there were no statements in the function.

Some example statement transformations follow.

`if`:

```
if ( b == "hello" ) print n;
```

translates to

```
( If ( Equal b "hello" ) ( Print n ) )
```

`while`:

```
while ( i != "10" ) { print i; i = i+"1"; }
```

translates to

```
( While ( NotEqual i "10" ) ( SLIST ( Print i ) ( = i ( Add i "1" ) ) ) )
```

Expressions have the operator at the root and the operands as children. For example,

```
i = a + "3" * "9" + f("jim");
```

would be translated to:

```
( = i ( Add ( Add a ( Mul "3" "9" ) ) ( f "jim" ) ) )
```

which looks like:

```
        =
        |
        v
        i -> Add
              |
              v
             Add --------------> f
              |                  |
              v                  v
              a -> Mul         "jim"
                    |
                    v
                   "3" -> "9"
```

5. Code generation from intermediate form

To translate the intermediate form (IF) to the stack code described above, we present the following simple little code generator. We use no devious C programming to make this fast or nifty because this document concentrates on how PCCTS grammars are developed not on how to write spectacular C code.

5.1. Modification of tree construction grammar

To convert from the translator above that printed out trees in lisp format, we add the following definitions needed as arguments to `gen()` in the `#header` action.

```
#define DONT_CARE         0
#define SIDE_EFFECTS      1
#define VALUE             2
```

Also, we replace all calls to `lisp(`*tree*`)` with `gen(`*tree*`, DONT_CARE)`.

5.2. Code generator

Function `gen()` presented below accepts an AST as an argument and walks the tree generating output when necessary. You will notice all the header information needed by this file, `code.c`. This illustrates one of the minor inconveniences of PCCTS. Any C file that needs to access PCCTS variables, rules, etc... must include all of the things that PCCTS generates at the head of all PCCTS generated files.

The code generator accepts an evaluation mode argument so that it can remove some unnecessary code. For example,

```
main()
{
    "3" + f();
}
f()
{}
```

would result in

```
#include "sc.h"
_main()
{
    BEGIN;
    CALL(f);
    POP;
    END;
}

f()
{
    BEGIN;
    END;
}
```

Code to push and add `"3"` was eliminated because it had no side effects. Only function calls have side effects in *sc*.

```
/*
 * Simple code generator for PCCTS 1.0x tutorial
 *
 * Terence Parr
 * Purdue University
 * Electrical Engineering
 * March 19, 1992
 */
#include <stdio.h>
#include "sym.h"
#include "charbuf.h"
#define AST_FIELDS int token, level, offset; char str[D_TextSize];
#define zzcr_ast(ast,attr,tok,text) create_ast(ast,attr,tok,text)
#include "antlr.h"
#include "ast.h"
#include "tokens.h"
#include "dlgdef.h"
#include "mode.h"
#define GLOBAL              0
#define PARAMETER           1
#define LOCAL               2
```

```
static int LabelNum=0, GenHeader=0;

#define DONT_CARE           0
#define SIDE_EFFECTS     1
#define VALUE               2


gen(t,emode)
AST *t;
int emode;  /* evaluation mode member { SIDE_EFFECTS, VALUE, DONT_CARE } */
{
    AST *func, *arg, *locals, *slist, *a, *opnd1, *opnd2, *lhs, *rhs;
    int n;


    if ( t == NULL ) return;
    if ( !GenHeader ) { printf("#include \"sc.h\"\n"); GenHeader=1; }

    switch ( t->token )
    {


    case DefineFunc :
        func = zzchild(t);
        arg = zzchild( zzsibling(func) );
        locals = zzchild( zzsibling( zzsibling(func) ) );
        slist = zzsibling( zzsibling( zzsibling(func) ) );
        if ( strcmp(func->str, "main") == 0 )
            printf("_%s()\n{\n\tBEGIN;\n", func->str);
        else
            printf("%s()\n{\n\tBEGIN;\n", func->str);
        for (a=locals; a!=NULL; a = zzsibling(a)) printf("\tLOCAL;\n");
        gen( slist, DONT_CARE );
        printf("\tEND;\n}\n");
        break;


    case SLIST :
        for (a=zzchild(t); a!=NULL; a = zzsibling(a))
        {
            gen( a, EvalMode(a->token) );
            if ( a->token == Assign ) printf("\tPOP;\n");
        }
        break;


    case DefineVar :
        printf("SCVAR %s;\n", zzchild(t)->str);
        break;
```

```
case Mul :
case Div :
case Add :
case Sub :
case Equal :
case NotEqual :
    opnd1 = zzchild(t);
    opnd2 = zzsibling( opnd1 );
    gen( opnd1, emode );
    gen( opnd2, emode );
    if ( emode == SIDE_EFFECTS ) break;
    switch ( t->token )
    {
    case Mul : printf("\tMUL;\n"); break;
    case Div : printf("\tDIV;\n"); break;
    case Add : printf("\tADD;\n"); break;
    case Sub : if ( opnd2==NULL ) printf("\tNEG;\n");
               else printf("\tSUB;\n");
               break;
    case Equal : printf("\tEQ;\n"); break;
    case NotEqual : printf("\tNEQ;\n"); break;
    }
    break;

case If :
    a = zzchild(t);
    gen( a, VALUE );              /* gen code for expr */
    n = LabelNum++;
    printf("\tBRF(iflabel%d);\n", n);
    a = zzsibling(a);
    gen( a, EvalMode(a->token) ); /* gen code for statement */
    printf("iflabel%d: ;\n", n);
    break;

case While :
    a = zzchild(t);
    n = LabelNum++;
    printf("wbegin%d: ;\n", n);
    gen( a, VALUE );              /* gen code for expr */
    printf("\tBRF(wend%d);\n", n);
    a = zzsibling(a);
    gen( a, EvalMode(a->token) ); /* gen code for statement */
    printf("\tBR(wbegin%d);\n", n);
    printf("wend%d: ;\n", n);
    break;

case Return :
    gen( zzchild(t), VALUE );
    printf("\tRETURN;\n");
    break;

case Print :
    gen( zzchild(t), VALUE );
    printf("\tPRINT;\n");
    break;
```

```
    case Assign :
        lhs = zzchild(t);
        rhs = zzsibling( lhs );
        gen( rhs, emode );
        printf("\tDUP;\n");
        if ( lhs->level == GLOBAL ) printf("\tSTORE(%s);\n", lhs->str);
        else printf("\tLSTORE(%d);\n", lhs->offset);
        break;

    case VAR :
        if ( emode == SIDE_EFFECTS ) break;
        if ( t->level == GLOBAL ) printf("\tPUSH(%s);\n", t->str);
        else printf("\tLPUSH(%d);\n", t->offset);
        break;

    case FUNC :
        gen( zzchild(t), VALUE );
        printf("\tCALL(%s);\n", t->str);
        break;

    case STRING :
        if ( emode == SIDE_EFFECTS ) break;
        printf("\tSPUSH(%s);\n", t->str);
        break;

    default :
        printf("Don’t know how to handle: %s\n", zztokens[t->token]);
        break;
    }
}

EvalMode( tok )
int tok;
{
    if ( tok == Assign || tok == If || tok == While ||
         tok == Print || tok == Return ) return VALUE;
    else return SIDE_EFFECTS;
}
```

5.3. Makefile for code generator

This makefile is roughly the same as before except that we need to tell PCCTS to generate tree information and we need to link in the code generator.

```
CFLAGS= -I../h
GRM=tut4
SRC=scan.c $(GRM).c err.c sym.c code.c
OBJ=scan.o $(GRM).o err.o sym.o code.o

tutorial: $(OBJ) $(SRC)
    cc -o $(GRM) $(OBJ)

$(GRM).c parser.dlg : $(GRM).g
    antlr -k 2 -gt $(GRM).g

scan.c : parser.dlg
    dlg -C2 parser.dlg scan.c
```

Table of Contents