Compiler Construction

Practical Exercise 3: Parser

Michael Engel

3.1 a Scanner

Complete the lex scanner specification in `src/scanner.l` so that it properly tokenizes VSL programs.

```
%{
#include <vslc.h>
%}
%option noyywrap
%option array
%option yylineno

WHITESPACE [\ \t\v\r\n]
COMMENT \/\/[^\n]+
QUOTED \"([^\"\n]|\\\")*\"
%%
{WHITESPACE}+           { /* Eliminate whitespace */ }
{COMMENT}               { /* Eliminate comments */ }
def                     { return FUNC; }
print                   { return PRINT; }
return                  { return RETURN; }
continue                { return CONTINUE; }
if                      { return IF; }
then                    { return THEN; }
else                    { return ELSE; }
while                   { return WHILE; }
do                      { return DO; }
begin                   { return OPENBLOCK; }
end                     { return CLOSEBLOCK; }
var                     { return VAR; }
[0-9]+                  { return NUMBER; }
[A-Za-z_][0-9A-Za-z_]*  { return IDENTIFIER; }
{QUOTED}                { return STRING; }
.                       { return yytext[0]; }
%%
```

3.1 a Scanner

What about these tokens?

<<

>>

:=

Detect in scanner vs. write sequence in grammar?

It's a matter of taste :)

```
%{
#include <vslc.h>
%}
%option noyywrap
%option array
%option yylineno

WHITESPACE [\ \t\v\r\n]
COMMENT \/\/[^\n]+
QUOTED \"([^\"\n]|\\\")*\"
%%
{WHITESPACE}+           { /* Eliminate whitespace */ }
{COMMENT}               { /* Eliminate comments */ }
def                     { return FUNC; }
print                   { return PRINT; }
return                  { return RETURN; }
continue                { return CONTINUE; }
if                      { return IF; }
then                    { return THEN; }
else                    { return ELSE; }
while                   { return WHILE; }
do                      { return DO; }
begin                   { return OPENBLOCK; }
end                     { return CLOSEBLOCK; }
var                     { return VAR; }
:=                      { return ASSIGN; }
\<\<                    { return LSHIFT; }
>>                      { return RSHIFT; }
[0-9]+                  { return NUMBER; }
[A-Za-z_][0-9A-Za-z_]*  { return IDENTIFIER; }
{QUOTED}                { return STRING; }
.                       { return yytext[0]; }
%%
```

3.1 b VSL Tree Construction

A node_t structure is defined in `include/ir.h`. Complete the auxiliary functions `node_init` and `node_finalize` so that they can initialize/free `node_t`-sized memory areas passed to them by their first argument. The function `destroy_subtree` should recursively remove the subtree below a given node, while `node_finalize` should only remove the memory associated with a single node.

Node construction functions:

```
%{
#include <vslc.h>

#define N0C(n,t,d) do { \
    node_init ( n = malloc(sizeof(node_t)), t, d, 0 ); \
} while ( false )
#define N1C(n,t,d,a) do { \
    node_init ( n = malloc(sizeof(node_t)), t, d, 1, a ); \
} while ( false )
#define N2C(n,t,d,a,b) do { \
    node_init ( n = malloc(sizeof(node_t)), t, d, 2, a, b ); \
} while ( false )
#define N3C(n,t,d,a,b,c) do { \
    node_init ( n = malloc(sizeof(node_t)), t, d, 3, a, b, c ); \
} while ( false )

%}
```

3.1 b VSL Tree Construction

Node init:

```
void
node_init (node_t *nd, node_index_t type, void *data, uint64_t n_children, ...)
{
    va_list child_list;
    *nd = (node_t) {
        .type = type,
        .data = data,
        .entry = NULL,
        .n_children = n_children,
        .children = (node_t **) malloc ( n_children * sizeof(node_t *) )
    };
    va_start ( child_list, n_children );
    for ( uint64_t i=0; i<n_children; i++ )
        nd->children[i] = va_arg ( child_list, node_t * );
    va_end ( child_list );
}
```

Destroy subtree:

```
void
destroy_subtree ( node_t *discard )
{
    if ( discard != NULL )
    {
        for ( uint64_t i=0; i<discard->n_children; i++ ) {
            destroy_subtree ( discard->children[i] );
        }
        node_finalize ( discard );
    }
}
```

3.1 b VSL Tree Construction

Node finalize:

```
void
node_finalize ( node_t *discard )
{
    if ( discard != NULL )
    {
        if ( discard->data != NULL )
        {
            free ( discard->data );
                   discard->data = NULL;
        }

        if ( discard->children != NULL )
        {
            free ( discard->children );
                   discard->children = NULL;
        }

        free ( discard );
        discard = NULL;
    }
}
```

3.1 b VSL Tree Construction

Node printing:

```
void
node_print ( node_t *root, int nesting )
{
    if ( root != NULL )
    {
        printf ( "%*c%s", nesting, ' ', node_string[root->type] );
        if ( root->type == IDENTIFIER_DATA ||
             root->type == STRING_DATA ||
             root->type == RELATION ||
             root->type == EXPRESSION )
            printf ( "(%s)", (char *) root->data );
        else if ( root->type == NUMBER_DATA )
            printf ( "(%lld)", *((int64_t *)root->data) );
        putchar ( '\n' );
        for ( int64_t i=0; i<root->n_children; i++ )
            node_print ( root->children[i], nesting+1 );
    }
    else
        printf ( "%*c%p\n", nesting, ' ', root );
}
```

3.1 c VSL Parser

Node construction convenience macros:

```
%{
#include <vslc.h>

#define N0C(n,t,d) do { \
    node_init ( n = malloc(sizeof(node_t)), t, d, 0 ); \
} while ( false )
#define N1C(n,t,d,a) do { \
    node_init ( n = malloc(sizeof(node_t)), t, d, 1, a ); \
} while ( false )
#define N2C(n,t,d,a,b) do { \
    node_init ( n = malloc(sizeof(node_t)), t, d, 2, a, b ); \
} while ( false )
#define N3C(n,t,d,a,b,c) do { \
    node_init ( n = malloc(sizeof(node_t)), t, d, 3, a, b, c ); \
} while ( false )

%}
```

Tokens and precedence:

```
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%token FUNC PRINT RETURN CONTINUE IF THEN ELSE WHILE DO OPENBLOCK CLOSEBLOCK
%token VAR NUMBER IDENTIFIER STRING ASSIGN LSHIFT RSHIFT
```

3.1 c VSL Parser

Rules:

```
%%
program :
      global_list { N1C ( root, PROGRAM, NULL, $1 ); }
    ;
global_list :
      global { N1C ( $$, GLOBAL_LIST, NULL, $1 ); }
    | global_list global { N2C ( $$, GLOBAL_LIST, NULL, $1, $2 ); }
    ;
global:
      function { N1C ( $$, GLOBAL, NULL, $1 ); }
    | declaration { N1C ( $$, GLOBAL, NULL, $1 ); }
    ;
statement_list :
      statement { N1C ( $$, STATEMENT_LIST, NULL, $1 ); }
    | statement_list statement { N2C ( $$, STATEMENT_LIST, NULL, $1, $2 ); }
    ;
print_list :
      print_item { N1C ( $$, PRINT_LIST, NULL, $1 ); }
    | print_list ',' print_item { N2C ( $$, PRINT_LIST, NULL, $1, $3 ); }
    ;
expression_list :
      expression { N1C ( $$, EXPRESSION_LIST, NULL, $1 ); }
    | expression_list ',' expression { N2C($$, EXPRESSION_LIST, NULL, $1, $3); }
    ;
variable_list :
      identifier { N1C ( $$, VARIABLE_LIST, NULL, $1 ); }
    | variable_list ',' identifier { N2C ( $$, VARIABLE_LIST, NULL, $1, $3 ); }
    ;
```

3.1 c VSL Parser

Rules:

```
argument_list :
      expression_list { N1C ( $$, ARGUMENT_LIST, NULL, $1 ); }
    | /* epsilon */ { $$ = NULL; }
    ;
parameter_list :
      variable_list { N1C ( $$, PARAMETER_LIST, NULL, $1 ); }
    | /* epsilon */ { $$ = NULL; }
    ;
declaration_list :
      declaration { N1C ( $$, DECLARATION_LIST, NULL, $1 ); }
    | declaration_list declaration { N2C ($$, DECLARATION_LIST, NULL, $1, $2); }
    ;
function :
      FUNC identifier '(' parameter_list ')' statement
        { N3C ( $$, FUNCTION, NULL, $2, $4, $6 ); }
    ;
statement :
      assignment_statement { N1C ( $$, STATEMENT, NULL, $1 ); }
    | return_statement { N1C ( $$, STATEMENT, NULL, $1 ); }
    | print_statement { N1C ( $$, STATEMENT, NULL, $1 ); }
    | if_statement { N1C ( $$, STATEMENT, NULL, $1 ); }
    | while_statement { N1C ( $$, STATEMENT, NULL, $1 ); }
    | null_statement { N1C ( $$, STATEMENT, NULL, $1 ); }
    | block { N1C ( $$, STATEMENT, NULL, $1 ); }
    ;
block :
      OPENBLOCK declaration_list statement_list CLOSEBLOCK
        { N2C ($$, BLOCK, NULL, $2, $3); }
    | OPENBLOCK statement_list CLOSEBLOCK { N1C ($$, BLOCK, NULL, $2 ); }
    ;
```

3.1 c VSL Parser

Rules:

```
assignment_statement :
      identifier ASSIGN expression
        { N2C ( $$, ASSIGNMENT_STATEMENT, NULL, $1, $3 ); }
    ;
return_statement :
      RETURN expression
        { N1C ( $$, RETURN_STATEMENT, NULL, $2 ); }
    ;
print_statement :
      PRINT print_list
        { N1C ( $$, PRINT_STATEMENT, NULL, $2 ); }
    ;
null_statement :
      CONTINUE
        { N0C ( $$, NULL_STATEMENT, NULL ); }
    ;
if_statement :
      IF relation THEN statement
        { N2C ( $$, IF_STATEMENT, NULL, $2, $4 ); }
    | IF relation THEN statement ELSE statement
        { N3C ( $$, IF_STATEMENT, NULL, $2, $4, $6 ); }
    ;
while_statement :
      WHILE relation DO statement
        { N2C ( $$, WHILE_STATEMENT, NULL, $2, $4 ); }
    ;
```

3.1 c

Rules:

```
expression :
      expression '+' expression
        { N2C ( $$, EXPRESSION, strdup("+"), $1, $3 ); }
    | expression '-' expression
        { N2C ( $$, EXPRESSION, strdup("-"), $1, $3 ); }
    | expression '*' expression
        { N2C ( $$, EXPRESSION, strdup("*"), $1, $3 ); }
    | expression '/' expression
        { N2C ( $$, EXPRESSION, strdup("/"), $1, $3 ); }
    | expression '|' expression
        { N2C ( $$, EXPRESSION, strdup("|"), $1, $3 ); }
    | expression LSHIFT expression
        { N2C ( $$, EXPRESSION, strdup("<<"), $1, $3 ); }
    | expression RSHIFT expression
        { N2C ( $$, EXPRESSION, strdup(">>"), $1, $3 ); }
    | '-' expression %prec UMINUS
        { N1C ( $$, EXPRESSION, strdup("-"), $2 ); }
    | '(' expression ')' { $$ = $2; }
    | number { N1C ( $$, EXPRESSION, NULL, $1 ); }
    | identifier
        { N1C ( $$, EXPRESSION, NULL, $1 ); }
    | identifier '(' argument_list ')'
        { N2C ( $$, EXPRESSION, NULL, $1, $3 ); }
    ;
declaration :
      VAR variable_list { N1C ( $$, DECLARATION, NULL, $2 ); }
    ;
print_item :
      expression
        { N1C ( $$, PRINT_ITEM, NULL, $1 ); }
    | string
        { N1C ( $$, PRINT_ITEM, NULL, $1 ); }
    ;
```

3.1 c

Rules:

```
identifier: IDENTIFIER { N0C($$, IDENTIFIER_DATA, strdup(yytext) ); }
number: NUMBER
      {
        int64_t *value = malloc ( sizeof(int64_t) );
        *value = strtol ( yytext, NULL, 10 );
        N0C($$, NUMBER_DATA, value );
      }
string: STRING { N0C($$, STRING_DATA, strdup(yytext) ); }
%%
```

The rest:

```
int
yyerror ( const char *error )
{
    fprintf ( stderr, "%s on line %d\n", error, yylineno );
    exit ( EXIT_FAILURE );
}
```

vslc.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <vslc.h>

node_t *root;

int yyparse(void);

int
main ( int argc, char **argv )
{
    yyparse();
    simplify_tree ( &root, root );
    node_print ( root, 0 );
    destroy_subtree ( root );
}
```

3.1 c VSL Parser

simplify.vsl:

```
var a,b
var c,d
func simplify ( e, f )
begin
    a := - ( 5*5 + 20/4 - 3 + 2 )
    print "a=", a
    if a=0 then b := simplify ( c, d )
    return 0
end
```

Output:

```
PROGRAM
GLOBAL_LIST
 DECLARATION
  VARIABLE_LIST
   IDENTIFIER_DATA(a)
   IDENTIFIER_DATA(b)
 DECLARATION
  VARIABLE_LIST
   IDENTIFIER_DATA(c)
   IDENTIFIER_DATA(d)
 FUNCTION
  IDENTIFIER_DATA(simplify)
  VARIABLE_LIST
   IDENTIFIER_DATA(e)
   IDENTIFIER_DATA(f)
  BLOCK
   STATEMENT_LIST
    ASSIGNMENT_STATEMENT
     IDENTIFIER_DATA(a)
     NUMBER_DATA(-29)
    PRINT_STATEMENT
     STRING_DATA("a=")
     IDENTIFIER_DATA(a)
    IF_STATEMENT
     RELATION(=)
      IDENTIFIER_DATA(a)
      NUMBER_DATA(0)
     ASSIGNMENT_STATEMENT
      IDENTIFIER_DATA(b)
      EXPRESSION((null))
       IDENTIFIER_DATA(simplify)
       EXPRESSION_LIST
        IDENTIFIER_DATA(c)
        IDENTIFIER_DATA(d)
    RETURN_STATEMENT
     NUMBER_DATA(0)
```