“Лабораторная работа 3.2 «Форматтер исходных текстов»”

9 сентября 2024 г.

Сергей Виленский, ИУ9-62Б

Цель работы

Целью данной работы является приобретение навыков использования генератора синтаксических анализаторов bison.

Индивидуальный вариант

‹индивидуальный вариант›

Реализация

lexer.l.c

```
%option reentrant noyywrap bison-bridge bison-locations caseless
%option extra-type="struct Extra *"

/* Подавление предупреждений для -Wall */
%option noinput nounput

%{

#include <stdio.h>
#include <stdlib.h>
#include "lexer.l.h"
#include "parser.h"  /* файл генерируется Bison’ом */

#define YY_USER_ACTION \
  { \
    int i; \
    struct Extra *extra = yyextra; \
    if (! extra->continued ) { \
      yylloc->first_line = extra->cur_line; \
      yylloc->first_column = extra->cur_column; \
    } \
    extra->continued = false; \
    for (i = 0; i < yyleng; ++i) { \
      if (yytext[i] == '\n') { \
        extra->cur_line += 1; \
        extra->cur_column = 1; \
      } else { \
        extra->cur_column += 1; \
      } \
    } \
    yylloc->last_line = extra->cur_line; \
    yylloc->last_column = extra->cur_column; \
  }

void yyerror(YYLTYPE *loc, yyscan_t scanner, const char *message) {
          printf("Error  (%d,%d):  %s\n",  loc->first_line,  loc-
>first_column, message);
}

%}

%%

\(\*([^\*]|\*[^\)])*\*\)|\{[^\}]*\}  {
    asprintf(&yylval->comment, "%s", yytext);
    return COMMENT;
}
\n  return NEW_LINE;

[\r\t ]+

\+      return '+';
\-      return '-';
\(      return LEFT_PAREN;
\)      return RIGHT_PAREN;
\.      return DOT;
\,      return COMMA;
\^      return CARET;
\;      return SEMICOLON;
\:      return COLON;
\=      return ASSIGN;

INTEGER     return KW_INTEGER;
BOOLEAN     return KW_BOOLEAN;
```

```
REAL        return KW_REAL;
CHAR        return KW_CHAR;
TEXT        return KW_TEXT;
PACKED      return KW_PACKED;
ARRAY       return KW_ARRAY;
OF          return KW_OF;
FILE        return KW_FILE;
SET         return KW_SET;
RECORD      return KW_RECORD;
END         return KW_END;
CASE        return KW_CASE;
CONST       return KW_CONST;
TYPE        return KW_TYPE;

[a-zA-Z][a-zA-Z0-9]*  {
    asprintf(&yylval->identifier, "%s", yytext);
    return IDENTIFIER;
}

[0-9]+(\.[0-9]+)?(E[\+\-]?[0-9]+)?  {
    yylval->unsigned_number = atof(yytext);
    return UNSINGNED_NUMBER;
}

'[^\']+'  {
    asprintf(&yylval->char_sequence, "%s", yytext);
    return CHAR_SEQUENCE;
}

%%

void init_scanner(FILE *input, yyscan_t *scanner, struct Extra *extra) {
    extra->continued = false;
    extra->cur_line = 1;
    extra->cur_column = 1;

    yylex_init(scanner);
    yylex_init_extra(extra, scanner);
    yyset_in(input, *scanner);
}

void destroy_scanner(yyscan_t scanner) {
    yylex_destroy(scanner);
}
```

lexer.l.h

```
#ifndef LEXER_H
#define LEXER_H

#include <stdbool.h>
#include <stdio.h>

#ifndef YY_TYPEDEF_YY_SCANNER_T
#define YY_TYPEDEF_YY_SCANNER_T
typedef void *yyscan_t;
#endif /* YY_TYPEDEF_YY_SCANNER_T */

struct Extra {
    bool continued;
    int cur_line;
    int cur_column;
};

struct Lines {
    char** lines;
    size_t count_lines;
};

void init_scanner(FILE *input, yyscan_t *scanner, struct Extra *extra);
void destroy_scanner(yyscan_t);

#endif /* LEXER_H */
```

parser.y.c

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "lexer.l.h"

struct Lines init_from_str(char* str) {
    struct Lines result;
    result.lines = (char**)malloc(sizeof(char*));
    result.count_lines = 1;

    asprintf(&result.lines[0], "%s", str);
    return result;
}
```

```
struct Lines union_lines(const struct Lines first, const struct Lines second) {
    if (first.count_lines == 0) {
        return second;
    }
    if (second.count_lines == 0) {
        return first;
    }

    struct Lines result;

   result.count_lines = first.count_lines + second.count_lines - 1;
   result.lines = (char**)malloc(result.count_lines * sizeof(char*));

    size_t i = 0;
    while (i != first.count_lines) {
        asprintf(&result.lines[i], "%s", first.lines[i]);
        ++i;
    }
       asprintf(&result.lines[i - 1], "%s%s", result.lines[i -
1], second.lines[0]);
    while (i + 1 - first.count_lines != second.count_lines) {
           asprintf(&result.lines[i], "%s", second.lines[i + 1 -
first.count_lines]);
        ++i;
    }

    free(first.lines);
    free(second.lines);

    return result;
}

struct Lines add_indents(const struct Lines lines, size_t start, size_t stop) {
    if (start + stop >= lines.count_lines) {
      return lines;
    }

    size_t i = start;
    while (i < lines.count_lines - stop - 1) {
        asprintf(&lines.lines[i], "  %s", lines.lines[i]);
        ++i;
    }

    if (
        lines.count_lines - stop - 1 >= start &&
        strlen(lines.lines[lines.count_lines - stop - 1]) != 0
    ) {
        asprintf(
          &lines.lines[lines.count_lines - stop - 1],
          "  %s", lines.lines[lines.count_lines - stop - 1]);
    }

    return lines;
}

void print_lines(const struct Lines lines) {
    size_t i = 0;
    while (i != lines.count_lines) {
        printf("%s\n", lines.lines[i]);
        ++i;
    }
}

%}

%define api.pure
%locations
%lex-param {yyscan_t scanner}  /* параметр для yylex() */
/* параметры для yyparse() */
%parse-param {yyscan_t scanner}

%union {
    char* identifier;
    float unsigned_number;
    char* char_sequence;
    char* comment;

    struct Lines lines;
}

%token NEW_LINE
%token LEFT_PAREN RIGHT_PAREN DOT COMMA CARET SEMICOLON COLON ASSIGN
%token KW_INTEGER KW_BOOLEAN KW_REAL KW_CHAR KW_TEXT
%token KW_PACKED KW_ARRAY KW_OF KW_FILE KW_SET KW_RECORD KW_END KW_CASE KW_CONST KW_TYPE

%token <identifier> IDENTIFIER
%token <unsigned_number> UNSINGNED_NUMBER
%token <char_sequence> CHAR_SEQUENCE
%token <comment> COMMENT

%type <lines> block
%type <lines> block_const_sequence
```

```
%type <lines> block_const
%type <lines> block_type_sequence
%type <lines> block_type

%type <lines> constant
%type <lines> unar_sign
%type <lines> constant_identifier

%type <lines> type
%type <lines> type_after_packed
%type <lines> simple_type_list

%type <lines> simple_type
%type <lines> type_identifier
%type <lines> common_type_identifier
%type <lines> identifier_list

%type <lines> field_list
%type <lines> identifier_with_type_list
%type <lines> identifier_with_type_seq
%type <lines> identifier_with_type
%type <lines> case_block
%type <lines> case_variant_sequence
%type <lines> case_variant
%type <lines> constant_list

%type <lines> space

%{
int yylex(YYSTYPE *yylval_param, YYLTYPE *yylloc_param, yyscan_t scanner);
void yyerror(YYLTYPE *loc, yyscan_t scanner, const char *message);
%}

%%

program:
      block
      {
        print_lines($1);
      }
    ;

block:
      block KW_CONST space block_const_sequence
      {
        $$ = union_lines(
            union_lines($1, init_from_str("CONST ")),
            union_lines($3, add_indents($4, 0, 0))
        );
      }
    | block KW_TYPE space block_type_sequence
      {
        $$ = union_lines(
            union_lines($1, init_from_str("TYPE ")),
            union_lines($3, add_indents($4, 0, 0))
        );
      }
    | space
      { $$ = $1; }
    ;
block_const_sequence:
      block_const
      { $$ = $1; }
    | block_const block_const_sequence
      { $$ = union_lines($1, $2); }
    ;
block_const:
      IDENTIFIER space ASSIGN space constant SEMICOLON space
      {
        $$ = union_lines(
            union_lines(
                union_lines(init_from_str($1), $2),
                union_lines(init_from_str(" = "), $4)
            ),
            union_lines(union_lines($5, init_from_str("; ")), $7)
        );
      }
    ;
block_type_sequence:
      block_type
      { $$ = $1; }
    | block_type block_type_sequence
      { $$ = union_lines($1, $2); }
    ;
block_type:
      IDENTIFIER space ASSIGN space type SEMICOLON space
      {
        $$ = union_lines(
            union_lines(
                union_lines(init_from_str($1), $2),
                union_lines(init_from_str(" = "), $4)
            ),
            union_lines(union_lines(
              $5,
              init_from_str("; ")
            ), $7)
        );
      }
    ;
    ;

constant:
      unar_sign constant_identifier
      { $$ = union_lines($1, $2); }
    | constant_identifier
      { $$ = $1; }
    | unar_sign UNSINGNED_NUMBER space
      {
        char buffer[50];
        snprintf(buffer, sizeof(buffer), "%g", $2);
      $$ = union_lines(union_lines($1, init_from_str(buffer)), $3);
      }
    | UNSINGNED_NUMBER space
      {
        char buffer[50];
        snprintf(buffer, 50, "%g", $1);
        $$ = union_lines(init_from_str(buffer), $2);
      }
    | CHAR_SEQUENCE space
      { $$ = union_lines(init_from_str($CHAR_SEQUENCE), $2); }
unar_sign:
      '+' space
      { $$ = union_lines(init_from_str("+"), $2); }
    | '-' space
      { $$ = union_lines(init_from_str("-"), $2); }
    ;
constant_identifier:
      IDENTIFIER space
      { $$ = union_lines(init_from_str($1), $2); }
    ;

type:
      simple_type
      { $$ = add_indents($1, 1, 0); }
    | CARET space type_identifier
    { $$ = add_indents(union_lines(union_lines(init_from_str("^"), $2), $3), 1, 0); }
    | KW_PACKED space type_after_packed
    { $$ = add_indents(union_lines(union_lines(init_from_str("PACKED "), $2), $3), 1, 0); }
    | type_after_packed
      { $$ = $1; }
    ;
type_after_packed:
      KW_ARRAY space simple_type_list KW_OF space type
      {
        $$ = add_indents(union_lines(
            union_lines(
                union_lines(init_from_str("ARRAY "), $2),
                union_lines($3, init_from_str(" OF "))
            ),
            union_lines($5, $6)
        ), 1, 0);
      }
    | KW_FILE space KW_OF space type
      {
        $$ = add_indents(union_lines(
        union_lines(union_lines(init_from_str("FILE "), $2), init_from_str("OF ")),
            union_lines($4, $5)
        ), 1, 0);
      }
    | KW_SET space KW_OF space simple_type
      {
        $$ = add_indents(union_lines(
        union_lines(union_lines(init_from_str("SET "), $2), init_from_str("OF ")),
            union_lines($4, $5)
        ), 1, 0);
      }
    | KW_RECORD space field_list KW_END space
      {
        $$ = union_lines(
            add_indents(
         union_lines(union_lines(init_from_str("RECORD "), $2), $3),
              1, 0
            ),
            union_lines(init_from_str("END"), $5)
        );
      }
    ;
simple_type_list:
      simple_type
      { $$ = $1; }
    | simple_type COMMA space simple_type_list
      {
        $$ = union_lines(
            union_lines($1, init_from_str(", ")),
            union_lines($3, $4)
        );
      }
    ;

simple_type:
      type_identifier
      { $$ = $1; }
    | LEFT_PAREN space identifier_list RIGHT_PAREN space
      {
        $$ = union_lines(
              add_indents(
                union_lines(
                  union_lines(init_from_str("("), $2),
                  $3
                ), 1, 0
              ),
              union_lines(init_from_str(")"), $5)
        );
      }
    | constant DOT space DOT space constant
      {
        $$ = union_lines(
            union_lines(
                union_lines($1, init_from_str(".")),
                union_lines($3, init_from_str("."))
            ),
            union_lines($5, $6)
        );
      }
    ;
type_identifier:
      common_type_identifier
      { $$ = $1; }
    | IDENTIFIER space
      { $$ = union_lines(init_from_str($1), $2); }
    ;
common_type_identifier:
      KW_INTEGER space
      { $$ = union_lines(init_from_str("INTEGER"), $2); }
    | KW_BOOLEAN space
      { $$ = union_lines(init_from_str("BOOLEAN"), $2); }
    | KW_REAL space
      { $$ = union_lines(init_from_str("REAL"), $2); }
    | KW_CHAR space
      { $$ = union_lines(init_from_str("CHAR"), $2); }
    | KW_TEXT space
      { $$ = union_lines(init_from_str("TEXT"), $2); }
    ;
identifier_list:
      IDENTIFIER space
      { $$ = union_lines(init_from_str($1), $2); }
    | IDENTIFIER space COMMA space identifier_list
      {
        $$ = union_lines(
        union_lines(union_lines(init_from_str($1), $2), init_from_str(", ")),
            union_lines($4, $5)
        );
      }
    ;

field_list:
      identifier_with_type_list
      { $$ = $1; }
    | identifier_with_type_seq case_block
      { $$ = union_lines($1, $2); }
    ;
identifier_with_type_list:
      identifier_with_type
      { $$ = $1; }
   | identifier_with_type SEMICOLON space identifier_with_type_list
      {
        $$ = union_lines(
            union_lines($1, init_from_str("; ")),
            union_lines($3, $4)
        );
      }
    ;
identifier_with_type_seq:
      identifier_with_type SEMICOLON space
     { $$ = union_lines(union_lines($1, init_from_str("; ")), $3); }
    | identifier_with_type SEMICOLON space identifier_with_type_seq
    { $$ = union_lines(union_lines($1, init_from_str("; ")), union_lines($3, $4)); }
    ;
identifier_with_type:
      identifier_list COLON space type
    { $$ = union_lines(union_lines($1, init_from_str(" : ")), union_lines($3, $4)); }
    ;
case_block:
    KW_CASE space IDENTIFIER space COLON space type_identifier KW_OF space case_variant_sequenc
      {
        $$ = union_lines(
      union_lines(
        union_lines(
          union_lines(init_from_str("CASE "), $2),
          union_lines(init_from_str($3), $4)
        ),
        union_lines(init_from_str(" : "), $6)
      ),
      union_lines(
        union_lines($7, init_from_str(" OF ")),
        union_lines($9, $10)
      )
    );
   }
  ;
case_variant_sequence:
   case_variant
   { $$ = $1; }
  | case_variant SEMICOLON space case_variant_sequence
  { $$ = union_lines(union_lines($1, init_from_str("; ")), union_lines($3, $4)); }
  ;
case_variant:
  constant_list COLON space LEFT_PAREN space field_list RIGHT_PAREN space
   {
    $$ = union_lines(
      union_lines(
        union_lines($1, init_from_str(" : ")),
        $3
      ),
      union_lines(
       add_indents(
        union_lines(
          union_lines(init_from_str("("), $5),
          $6
        ), 1, 0
       ),
       union_lines(init_from_str(")"), $8)
      )
    );
   }
  ;
constant_list:
   constant
   { $$ = $1; }
  | constant COMMA space constant_list
   {
    $$ = union_lines(
            union_lines($1, init_from_str(", ")),
            union_lines($3, $4)
        );
      }
    ;

space:
      COMMENT space[TAIL]
      { $$ = union_lines(init_from_str($COMMENT), $TAIL); }
    | NEW_LINE space[TAIL]
      {
        $$.lines = (char**)malloc(2 * sizeof(char*));
        $$.lines[0] = $$.lines[1] = "";
        $$.count_lines = 2;

        $$ = union_lines($$, $TAIL);
      }
    |
      {
        $$.lines = NULL;
        $$.count_lines = 0;
      }
    ;

%%

int main(int argc, char *argv[]) {
    FILE *input = 0;
    yyscan_t scanner;
    struct Extra extra;

    if (argc > 1) {
        printf("Read file %s\n", argv[1]);
        input = fopen(argv[1], "r");
    } else {
        printf("No file in command line, use stdin\n");
        input = stdin;
    }

    init_scanner(input, &scanner, &extra);
    yyparse(scanner);
    destroy_scanner(scanner);

    if (input != stdin) {
        fclose(input);
    }
    return 0;
}
```

run.sh

```
#!/bin/bash

lex -o lexer.c lexer.l.c
yacc -d -o parser.c parser.y.c
gcc -o calc lexer.c parser.c

./calc $1

rm -f parser.c parser.h
rm -f lexer.c lexer.h
rm -f calc
```

Тестирование

Входные данные

```
Type {123}
  Coords = Record x, y: INTEGER end;
Const
  MaxPoints = 100;
type
  CoordsVector = array 1..MaxPoints of Coords;

const
  Heigh = 480;
  Width = 640;
  Lines = 24;
  Columns = 80;
type
  BaseColor = ( red, green,blue, highlited);
  Color = set of BaseColor;
  GraphicScreen = array 1..Heigh of array 1..Width of Color;
  TextScreen = array 1..Lines of array 1..Columns of
    record
      Symbol : CHAR;
      SymColor : Color;
      BackColor : Color
    end;
```

```
(* определения токенов }
{ определения токенов *)
(* определения токенов *)
{ определения токенов }
{ определения токенов *)
(* определения токенов }
TYPE
  Domain = (Ident, IntNumber, RealNumber);
  Token = record
    fragment : record
      start, following : record
        row, col : INTEGER
      end
    end;
    case tokType : Domain of
      Ident : (
        name : array 1..32 of CHAR
      );
      IntNumber : (
        intval : INTEGER
      );
      RealNumber : (
        realval : REAL
      )
  end;

  Year = 1900..2050;

  List = record
    value : Token;
    next : ^List
  end;
```

Вывод на stdout

```
Type {123}
  Coords = Record x, y: INTEGER end;
Const
  MaxPoints = 100;
type
  CoordsVector = array 1..MaxPoints of Coords;

const
  Heigh = 480;
  Width = 640;
  Lines = 24;
  Columns = 80;
```

```
type
  BaseColor = ( red, green,blue, highlited);
  Color = set of BaseColor;
  GraphicScreen = array 1..Heigh of array 1..Width of Color;
  TextScreen = array 1..Lines of array 1..Columns of
    record
      Symbol : CHAR;
      SymColor : Color;
      BackColor : Color
    end;

 (* определения токенов }
{ определения токенов *)
 (* определения токенов *)
 { определения токенов }
 { определения токенов *)
(* определения токенов }
TYPE
  Domain = (Ident, IntNumber, RealNumber);
  Token = record
    fragment : record
      start, following : record
        row, col : INTEGER
      end
    end;
    case tokType : Domain of
      Ident : (
        name : array 1..32 of CHAR
      );
      IntNumber : (
        intval : INTEGER
      );
      RealNumber : (
        realval : REAL
      )
  end;

  Year = 1900..2050;

  List = record
    value : Token;
    next : ^List
  end;
```

Вывод

В ходе выполнения данной работы были приобретены навыки использования генератора синтаксических анализаторов bison.