

Práctica 3: Intérprete para un lenguaje P

Iván López de Munain Quintana

Gramáticas y Lenguajes Formales

23 de mayo de 2019



Índice

1. Especificaciones léxicas	3
1.1. Identificadores y palabras claves	3
1.2. Constantes numéricas	3
1.3. Constantes carácter y tiras de caracteres	4
1.4. Separadores	4
1.5. Comentarios	5
1.6. Operadores	6
1.7. Errores	7
2. Especificaciones sintácticas	7
2.1. Asignación	7
2.2. Sentencia vacía	9
2.3. Funciones de entrada y salida	9
2.4. If-else	11
2.5. While	12
2.6. Funciones predefinidas	13
2.7. Expresiones	13
3. Extensiones	14
3.1. For	14
4. Ejecución	15

Los objetivos de esta práctica son afianzar los conceptos aprendidos sobre las herramientas *Lex* y *Yacc* para la construcción de analizadores sintácticos y léxicos. Básicamente se pretende construir un intérprete de un lenguaje (P) que es similar al lenguaje C. Cabe remarcar que a la hora de realizar los siguiente apartados he utilizado como ficheros base los realizados por el profesor Dusan Kolár ¹ en la exposición que realizó para el alumnado.

1. Especificaciones léxicas

En este apartado se va a describir los elementos léxicos del lenguaje P, todos ellos van a ser descritos mediante expresiones regulares en el fichero llamado *pcc.l*.

1.1. Identificadores y palabras claves

Los identificadores se definen mediante un conjunto de letras, números, '_' tal que comiencen por '_' o por una letra. Esto se consigue con la siguiente expresión regular.

```
LETTER    ([_a-zA-Z])
IDENT     ((" ")*{LETTER}({LETTER}|{DIGIT})*)
```

Siendo *IDENT* el correspondiente identificador.

1.2. Constantes numéricas

Este tipo de elementos poseen una parte entera que puede ir seguida de una parte decimal, además podrán expresarse en notación decimal (representándose el exponente mediante 'E' o 'e' seguido del signo '-' o '+'). Esto se consigue en el fichero *pcc.l* a través de las expresiones regulares siguientes:

```
DIGIT      ([0-9])
DIGITS     ({DIGIT}+)

EXP         ([eE] [-+]?{DIGITS})

FLOAT1      ({DIGITS}"."{DIGITS})
FLOAT2      ({DIGITS}{EXP})
FLOAT3      ({DIGITS}"."{DIGITS}{EXP})

FLOAT       ({FLOAT1}|{FLOAT2}|{FLOAT3})
```

¹Profesor de la Universidad de Brno (Chequia)

Posteriormente para realizar su almacenamiento en el formato pedido (coma flotante de doble precisión o double) bastará con implementar lo siguiente en la parte intermedia del archivo *Lex*.

```
{DIGITS}      {
    long int li;
    sscanf(yytext, "%ld", &li);
    yyFloat(yylval) = (double)li;
    yyFlag(yylval) = fFLOAT;

    return FLOAT;
}

{FLOAT}      {
    sscanf(yytext, "%lf", &( yyFloat(yylval) ));
    yyFlag(yylval) = fFLOAT;

    return FLOAT;
}
```

Figura 1: Acción al encontrarse constantes numéricas.

1.3. Constantes carácter y tiras de caracteres

Tan solo se han implementado las tiras de caracteres, no operaciones que las utilicen.

```
STRSTART      (["])

%%

{STRSTART}    {
    yyStr(yylval) = readStr();
    yyFlag(yylval) = fSTR;

    return STR;
}
```

1.4. Separadores

Para realizar este apartado tenemos que tener en cuenta que los espacios en blanco, tabuladores (*\t*), saltos de línea (*\n*), salto de carro (*\r*) y form feed (*\f*).

```
EOL          ([\n])

WSPC          ([\t\f\r])
WSPCS         ({WSPC}+)
```

Como en estas ocasiones no queremos que el intérprete haga nada, simplemente les ignore tan solo tendremos que añadir lo siguiente en la parte intermedia de *pcc.l*:

```
{EOL}          {
                ++yylineno;
                return EOL;
            }

{WSPC}          ; /* nada que hacer, espacio en blanco */
```

Cabe destacar que para indicar los bloques de las secciones a través de llaves se hace de la misma forma.

1.5. Comentarios

Pese a que en el enunciado de la práctica dice que tan solo hay que implementar un intérprete que ignore los comentarios de una línea (no permitiendo los multilínea), voy a implementar que ignore los multilínea pues el otro caso es un caso específico de este. Para el caso de comentario multilínea cada vez que se encuentre el patrón `"/*` se ejecutará el método *IgnoraComentarioMultiLinea*(*nlin*, *ncol*):

```
"/\*" IgnoraComentarioMultiLinea(&nlin, &ncol);
```

Siendo el código de dicho método el siguiente:

```
void IgnoraComentarioMultiLinea(int *nlin, int *nc)
{
    int t, c;
    int done = 0;

    // DBG:
#ifdef VERBOSE
    fprintf(stderr, "___DBG: Procesando comentario (%d, %d)-{", *nlin, *nc);
#endif
    while(!done) {
        switch(t=input())
        {
            /* EOF: Bad comment */
            case EOF:
                fprintf(stderr, "___ERROR: unclosed comment, expect */\n");
                done = 1;
                break;

            case '*':
                if((c = input()) == '/') { *nc = *nc + 2; done = 1; }
                else { *nc = *nc + 1; unput(c); }
#ifdef VERBOSE
                if (!done) { fputc(t, stderr); fputc(c, stderr); }
#endif
                break;

            default:
                if (t=='\n') { *nlin = *nlin + 1; *nc = 1; }
                else { *nc = *nc + 1; }
#ifdef VERBOSE
                fputc(t, stderr);
#endif
                break;
        }
    }
#ifdef VERBOSE
    fprintf(stderr, ")-(%d, %d)\n", *nlin, *nc);
#endif
}
```

Figura 2: Código del método IgnoraComentarioMultiLinea.

Cabe remarcar que he optado por esta solución porque la considero más completa que aquella que solo contemple comentarios de una línea. En caso de implementar esa solución bastaría con que cada vez que el analizador se encontrase un patrón `"/"` se ejecutase el método *IgnoraComentario*(*nlin*, *ncol*) e ignorar todo aquello que se encontrase hasta el primer salto de línea (`\n`). En el caso de ignorar el comentario multilínea se ignora hasta encontrar el patrón `"*/"`.

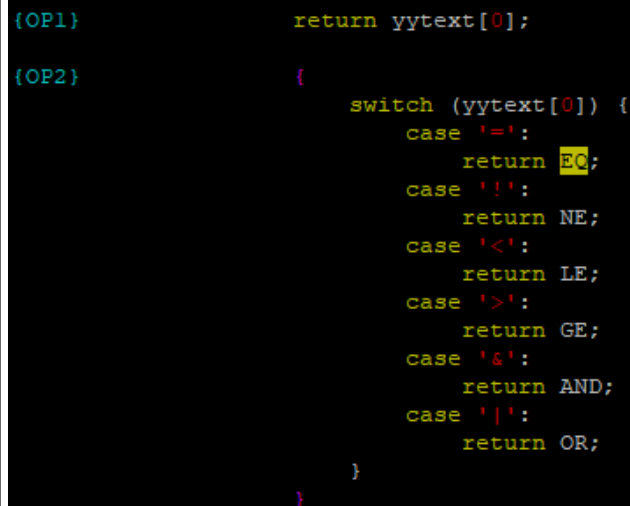
1.6. Operadores

- **Aritméticos:** `'+'` (suma), `'-'` (resta, cambio de signo), `'*'` (producto), `'/'` (división), `'%'` (división entera), `'^'` (potencia).
- **Lógicos:** `'>'` (mayor), `'<'` (menor), `'>='` (mayor o igual), `'<='` (menor o igual), `'=='` (igual), `'!='` (diferente), `'!'` (not), `'&&'` (and), `'||'` (or).
- **Asignación:** `'='` que se empleará en las sentencias de asignación.

Para representar dichos elementos hay que añadir en las definiciones del *lexer* lo siguiente:

```
OP1      ([ -+*/=<>?: () ! ^ ] )
OP2      ("==" | "!=" | "<=" | ">=" | "&&" | "||" | ")
```

Como podemos ver he realizado dos distinciones entre todas las operaciones dando lugar a dos comportamientos distintos en función si se encuentra el patrón *OP1* o *OP2*.



```
{OP1}      return yytext[0];
{OP2}      {
            switch (yytext[0]) {
                case '=':
                    return EQ;
                case '!':
                    return NE;
                case '<':
                    return LE;
                case '>':
                    return GE;
                case '&':
                    return AND;
                case '|':
                    return OR;
            }
        }
```

Figura 3: Acciones en función de las operaciones.

1.7. Errores

Como bien se indica al comienzo del enunciado, en caso de que el intérprete lea cualquier carácter que no se corresponda con las reglas expuestas anteriormente será señalado como un error sintáctico y se indicará su localización. Es por esto que en la parte intermedia del archivo *Lex* se definirá el siguiente patrón (usando el punto porque representa cualquier caracter, se dispone la última para así solo activarse una vez que ninguno de los demás patrones se haya disparado):

```
. {  
    prError(yylineno,"Carácter inesperado en la entrada: %c [%d]\n",  
    yytext[0],yytext[0],NULL);  
}
```

2. Especificaciones sintácticas

En esta parte de la práctica, de todas las especificaciones requeridas, he conseguido implementar las siguientes:

- **Asignación.**
- **Sentencia vacía.**
- **Sentencia bloque.**
- **Funciones de entrada y salida: read y write.**
- **Sentencia condicional if/else.**
- **Bucle condicional while.**

2.1. Asignación

Para que una asignación sea reconocida por el intérprete del lenguaje *P* tiene que constar de un identificador en la parte izquierda de la igualdad ('=') y una expresión en la parte derecha. Obviamente puede ir seguido de cero o más caracteres de separación. Para conseguir esto es necesario añadir lo siguiente en el archivo *pcc.y*:

```
%token <s> IDENT PUNTOYCOMA  
%type <s> ternary  
  
%%  
  
statement  
: IDENT '=' ternary PUNTOYCOMA  
{  
    $$ .flag = FAST;  
    $$ .u.ast = mkNd('=', mkSlf(IDENT,$1.u.vStr), $3.u.ast);  
}
```

En donde tanto IDENT (mostrado ya anteriormente) como PUNTOYCOMA (";") están definidos en el analizador léxico. Cabe resaltar que el ';' no sería necesario definirlo y bastaría con ponerlo directamente entre comillas simples. En el lexer además tendríamos que definir la acción a realizar cada vez que se encuentren dichos patrones, en caso de punto y coma no hacer nada, mientras que si se encuentra un patrón concordante con IDENT sería lo siguiente:

```
{PUNTOYCOMA}      {
                    ++yylineno;
                    return PUNTOYCOMA;
                    }

{IDENT}            {
                    unsigned i = 0;
                    int r=-1;
                    char *res;

                    lower(yytext);

                    while (i<KWLEN && r<0) {
                        if ((r=strcmp(keywords[i],yytext))==0) return keycodes[i];
                        ++i;
                    }

                    yyStr(yylval)=sdup(yytext);
                    yyFlag(yylval)=fIDENT;

                    return IDENT;
                    }
```

Además de esto, las *reglas de producción* asociadas a *ternary* son expresiones (IDENT, SIN, TAN, FLOAT...). Dichas asociaciones y reglas de producción pueden verse en el archivo *pcc.y*.

En el fragmento de código mostrado anteriormente vemos también cómo se va construyendo el *árbol de sintaxis abstracta (AST)* en el que el nodo raíz es el '=', el hijo izquierdo es IDENT y el derecho *ternary*. Para acceder a los valores de dichos registros utilizamos el símbolo \$. Para ir construyendo dicho árbol incluimos en el fichero *astree.c* la parte correspondiente a '=':

```
case '=':
    insertModify( sv(left(root)), expr(right(root)) );
    break;
```


2.2. Sentencia vacía

Tal y como ha sido presentada la sentencia vacía en el enunciado entiendo el concepto como una sentencia que tan solo consta de ';'. Para implementar esto basta con añadir que los elementos de un programa pueden ser o bien sentencias seguidas de un salto de línea, o solo un salto de línea o solo un punto y coma. Esto es añadido en el fichero *yacc*.

2.3. Funciones de entrada y salida

2.3.1. Write

Esta función venía ya implementada por Dusan Kolár, tan solo requería cambiar de nombre de *print* a *write*. Este hecho me costó bastante porque no recordaba que los *Keywords* y los *Keycodes* tenían que estar ordenados alfabéticamente. A continuación se va a mostrar las correspondientes reglas de producción y su implementación para construir el *AST*:

```
| WRITE '(' ternary ')' PUNTOYCOMA
{
    $$ .flag = fAST;
    $$ .u.ast = mkNd(WRITE, NULL, $3.u.ast);
}
| WRITE '(' STR ')' PUNTOYCOMA
{
    $$ .flag = fAST;
    $$ .u.ast = mkNd(WRITE, mkSlf(STR, $3.u.vStr), NULL);
}
| WRITE '(' STR COMA ternary ')' PUNTOYCOMA
{
    $$ .flag = fAST;
    $$ .u.ast = mkNd(WRITE, mkSlf(STR, $3.u.vStr), $5.u.ast);
}
```

Figura 4: Dentro del fichero pcc.y.

```
case WRITE:
    if (left(root) == NULL) {
        printf("%g\n", expr(right(root)) );
    } else if (right(root) == NULL) {
        puts( sv(left(root)) );
    } else {
        printf("%s%g\n", sv(left(root)), expr(right(root)) );
    }
    break;
```

Figura 5: Dentro del fichero astree.c.

2.3.2. Read

Esta función de entrada puede tomar dos parámetros (separados por una coma) o solo uno. En caso de ser dos parámetros se tratará de un *String* y una variable que almacenará el valor leído. Otra posibilidad de ejecutar dicha función es sin texto adicional proporcionando un único parámetro al que se le quiere dar valor. Para conseguir esto bastaría con añadir la siguiente *regla de producción* para la variable *statement*:

```
| READ '(' IDENT ')' PUNTOYCOMA
{
    $$ .flag = fAST;
    $$ .u.ast = mkNd(READ, NULL, mkSlf(IDENT, $3.u.vStr));
}
| READ 'STR COMA IDENT' PUNTOYCOMA
{
    $$ .flag = fAST;
    $$ .u.ast = mkNd(READ, mkSlf(STR, $3.u.vStr), mkSlf(IDENT, $5.u.vStr));
}
```

Figura 6: Dentro del fichero pcc.y.

Al igual que en el caso de PUNTOYCOMA el token COMA podría haberse puesto literalmente entre comillas simples (en el lexer ha tenido que ser definido, tampoco se realiza nada cuando el intérprete lo lee). Podemos ver en ambos casos cómo se van construyendo el *AST*. A continuación se va a mostrar la parte del *switch* correspondiente, encontrándose éste en el fichero *astree.c* el cual es el encargado de construir el árbol:

```
case READ:
    if (left(root) == NULL) {
        double rval;
        scanf("%lf", &rval);
        insertModify(sv(right(root)), rval);
    } else {
        double rval;
        printf("%s", sv(left(root)));
        scanf("%lf", &rval);
        insertModify( sv(right(root)), rval);
    }
    break;
```

Figura 7: Funcionamiento de read.

Como vemos en la imagen superior, en caso de que el hijo izquierdo sea nulo significa que estamos en el caso de un único parámetro (no hay String) por lo que mediante las funciones *scanf* y *insertModify* se almacena el valor que se encuentra en el hijo derecho. En la situación de ser dos parámetros la única diferencia reside en que, además de lo ya comentado, se imprime el hijo izquierdo (el String).

2.4. If-else

La estructura del *if-else* implementada es la requerida en el enunciado de la práctica, siendo la regla de producción construida la siguiente:

```
ifstmt
: IF '(' ternary ')' CORCHETES EOL body EOL CORCHETES
{
  $$flag = fAST;
  $$u.ast = mkNd(IF, $3.u.ast,$7.u.ast);
}
| IF '(' ternary ')' CORCHETES EOL body EOL CORCHETES ELSE CORCHETES EOL body EOL CORCHETES
{
  $$flag = fAST;
  $$u.ast = mkNd(ELSE, $3.u.ast,mkNd(';', $7.u.ast,$13.u.ast));
}
;

body
: statement
{
  $$flag = fAST;
  $$u.ast = appR(';', NULL, $1.u.ast);
}
| body EOL statement
{
  $$flag = fAST;
  $$u.ast = appR(';', $1.u.ast, $3.u.ast);
}
;
```

En donde *ifstmt* se encuentra en las regla correspondiente a *statement*. Podemos observar cómo se definen dos reglas distintas, una para el caso *if* simple y otra para el caso *if-else*, además también es necesario definir un cuerpo (*body*) para que pueda haber más de una sentencia. La implementación correspondiente para construir el árbol es la siguiente:

```

case IF:
    if(expr(left(root))) {
        evaluate(right(root));
    }
    break;

case ELSE:
    if(expr(left(root))) {
        evaluate(left(right(root)));
    } else {
        evaluate(right(right(root)));
    }
    break;

```

Figura 8: Fichero astree.c

Se puede observar cómo en el caso del *if* solo se evalúa la expresión del hijo izquierdo, en caso de ser cierto se evalúan las sentencias del hijo derecho. En el caso de usar *else* necesitamos construir un nivel más del árbol para considerar la alternativa de evaluar las sentencias del cuerpo correspondiente al *else*.

2.5. While

Una vez implementado el *if-else* realizar esto sería bastante simple, la explicación al respecto es bastante similar por lo que no entraré muy en detalle en el código mostrado a continuación.

```

whilestmt
: WHILE '(' ternary ')' CORCHETES EOL  body EOL  CORCHETES
{
    $$ .flag = FAST;
    $$ .u.ast = mkNd(WHILE, $3.u.ast, $7.u.ast);
}
;

```

Al igual que en el caso anterior *whilestmt* es una regla de producción de *statement* y también tenemos que hacer uso de la variable *body*.

```

case WHILE:
    while(expr(left(root))) {
        if(right(root) != NULL) {
            evaluate(right(right(root)));
        }
    }
    break;

```

Figura 9: Fichero astree.c

2.6. Funciones predefinidas

En este apartado se quiere dar soporte tanto a funciones trigonométricas (*sin*, *cos*, *tan*, *asin*, *acos* y *atan*) como funciones matemáticas (*log*, *log10*, *exp*, *ceil* y *floor*). En esta situación la implementación de todas es análoga por lo que solo se va a mostrar el caso de una.

Para que el intérprete consiga reconocer el seno de un valor es necesario incluir en las *reglas de producción* de expresiones (*expr*) lo siguiente:

```
| SIN '(' ternary ')'  
  {  
    $$ . flag = fAST;  
    $$ . u . ast = mkNd(SIN, $3 . u . ast, NULL);  
  }
```

Podemos ver cómo se reconoce el término SIN y cómo se construye el árbol con el hijo *ternary*. Una vez reconocido este patrón es necesario implementar en *astree.c* el correspondiente caso dentro del switch tal que se devuelva el cálculo de aplicar la función seno a lo que esté conteniendo *ternary*.

```
case SIN:  
  return sin(expr(left(root))) );
```

Cabe recordar que SIN ha tenido que ser definida como término en *pcc.y* y además como palabra clave en el analizador léxico (al igual que el resto de palabras claves usadas hasta el momento: *read*, *cos*, *sin*, ...).

2.7. Expresiones

Lo referido a este apartado, tanto como las propiedades de asociación (por la izquierda, por la derecha...), como las reglas de producción, como la implementación para construir el árbol se ha tomado la solución dado por Dusan por lo que no se incluirán aquí.

3. Extensiones

De todas las extensiones que se proponían he decidido realizar solo una debido a que nos encontramos en la etapa final del curso y hay demasiada carga de trabajo del resto de asignaturas. He decidido implementar el *for*.

3.1. For

Cabe destacar que esta opción me ha costado más implementarla debida a que la sentencia cuenta con más elementos por lo que será necesario construir un árbol de mayor profundidad que en los casos anteriores. Antes de nada cabe destacar que mi implementación del *for* no sigue exactamente la misma sintaxis que la pedida en el enunciado, esto es debido a cómo había implementado las asignaciones con los puntos y coma. Para que el intérprete lo acepte tiene que tener el siguiente formato:

```
for(i=0;i=i+1;i<10){  
    // sentencias  
}
```

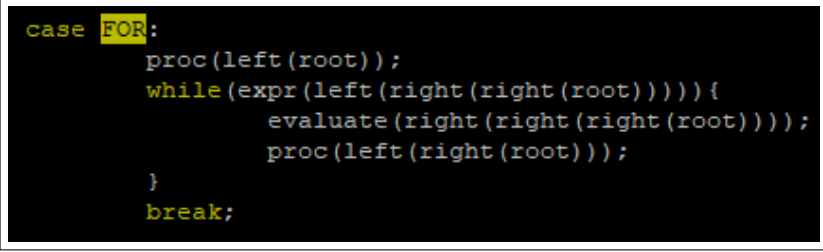
En lugar de lo requerido:

```
for(i=0;i<10;i=i+1){  
    // sentencias  
}
```

En el archivo *yacc* tendríamos que introducir la siguiente regla:

```
forstmt  
    : FOR '(' statement statement ternary ')' CORCHETES EOL body EOL CORCHETES  
    {  
        $$ . flag = FAST;  
        $$ . u . ast = mkNd(FOR,$3 . u . ast,mkNd(';', $4 . u . ast,mkNd(';', $5 . u . ast,$9 . u . ast)));  
    }  
    ;
```

Siendo *forstmt* cuerpo de la cabecera *statement*. Podemos observar cómo la construcción del *AST* es mucho más compleja debido a su profundidad. Para implementarlo incorporamos el código siguiente en el que se realiza la asignación de la variable que guía el bucle, se comprueba la condición del bucle y si se cumple se evalúa el cuerpo y posteriormente se actualiza el valor de la variable.



```
case FOR:  
    proc(left(root));  
    while(expr(left(right(right(root))))) {  
        evaluate(right(right(right(root))));  
        proc(left(right(root)));  
    }  
    break;
```

Figura 10: Fichero *astree.c*

4. Ejecución

Remarcar que solo existe un fichero de prueba en el que se realizan las comprobaciones para todas las funcionalidades implementadas en la práctica. Además de esto se ha realizado un fichero *bash* en el que se crea el intérprete que posteriormente habrá que ejecutar con el fichero de prueba llamado *test/ejemplo*. El contenido del *bash* es el siguiente:

```
bison --defines -v pcc.y -o pcc.c
flex -o inter05.lex.c pcc.l
gcc pcc.c inter05.lex.c symtab.c stduse.c astree.c -o x_pcc -lm
```

Referencias

- [1] Valentín Cardenoso Payo. *Documentación de la asignatura Gramáticas y Lenguajes Formales*. [UVA]
- [2] Diapositivas y conferencia de Dusan Kolár. [Universidad de Brno]