9.1 Introducción

El input de un programa de ordenador tiene generalmente alguna estructura; cada programa de ordenador que acepta input se puede pensar como definiendo un lenguaje de input que es aceptado por el programa. Un lenguaje de input puede ser tan complejo como un lenguaje de programación, o tan sencillo como una secuencia de números. Desafortunadamente, los dispositivos de input usuales son muy limitados, difíciles de usar, y a menudo carecen de verificación de validez de sus propios inputs.

El yacc proporciona una herramienta general para describir el input para un programa de ordenador. El nombre yacc quiere decir "yet another compiler-compiler". El usuario del yacc especifica las estructuras de su input, junto con el código que se va a llamar a medida que se reconocen cada estructura. Yacc convierte tal especificación en una subrutina que maneja el proceso de input; frecuentemente, es conveniente y apropiado hacer que esta subrutina maneje el flujo de control de la aplicación del usuario.

La subrutina producida por el yacc llama a una rutina proporcionada por el usuario para devolver el siguiente item de input básico. Por lo tanto, el usuario puede especificar su input en términos de caracteres de input individuales o en términos de construcciones de nivel más alto tales como nombres y números. La rutina proporcionada por el usuario puede también manejar características idiomáticas tales como convenciones de comentarios y de continuación, las cuales normalmente desafían las especificaciones gramaticales sencillas. La clase de especificaciones aceptadas es una clase muy general: gramáticas LALR con reglas que evitan la ambigüedad.

Además de los compiladores de C, APL, Pascal, RATFOR, etc, el yacc se ha usado también para lenguajes menos convencionales, incluyendo un lenguaje de fotocomposición, varios lenguajes de calculadoras de escritorio, un sistema de recuperación de documentos, y un sistema de depuración FORTRAN.

El yacc proporciona una herramienta general para imponer estructuras de input a un programa de ordenador. El usuario del yacc prepara una especificación del proceso de input; ésto incluye reglas que describen la estructura de input, código que se va a llamar cuando estas reglas sean reconocidas, y una rutina de bajo nivel para hacer el input básico. Entonces el yacc genera una función para controlar el proceso de input. Esta función, llamada parser, llama a la rutina de input de bajo nivel proporcionada por el usuario (llamada analizador léxico) para coger los items básicos (llamados tokens) del input. Estos tokens están organizados según las reglas de estructuras de input, llamadas reglas

gramáticas; cuando una de estas reglas ha sido reconocida, entonces se llama al código de usuario proporcionado por esta regla, en acción; las acciones tienen la habilidad de devolver valores y hacer uso de los valores de otras acciones.

El yacc está escrito en un dialecto portátil de C y las acciones, y las subrutina de output, están también en C. Lo que es más, muchas de las convecciones sintácticas del yacc siguen las del lenguaje C.

El corazón de la especificación de input es una colección de reglas gramáticas. Cada regla describe una estructura permisible y le da un nombre. Por ejemplo, una regla gramática puede ser:

fecha : nombre-mes dia ',' año;

Aquí, fecha, nombre-mes, día, y año representan estructuras de interés en el proceso de input; presumiblemente, nombre-mes, día y año están definidas en algún otro lugar. La coma (,) está incluida entre comillas simples; ésto implica que la coma va a aparecer literalmente en el input. El punto y coma y los dos puntos sirven simplemente como puntuación, y no tienen significado en el control del input. Por lo tanto, con las definiciones correctas, el input:

julio 4, 1776

puede coincidir con la regla anterior.

Una parte importante del proceso de input se lleva a cabo por medio del analizador léxico. Esta rutina de usuario lee el input reconociendo las estructuras de nivel bajo, y comunica estos tokens al parser. A las estructuras que el analizador léxico reconoce se les llama "símbolo terminal", mientras que la estructura que reconoce el parser se le llama "símbolo no terminal". Para evitar confusión, normalmente a los símbolo terminales se les conoce como tokens.

Hay un considerable retraso para decidir si reconocer estructuras usando el analizador léxico o usando las reglas gramaticales. Por ejemplo, las reglas

nombre-mes : 'E' 'n' 'e' ;

```
nombre-mes : 'F' 'e' 'b' ;
...
nombre-mes : 'D' 'i' 'c' ;
```

se podría usar en el ejemplo anterior. El analizador léxico sólo necesitaría reconocer letras individuales, y nombre-mes sería un símbolo no terminal. Tales reglas de bajo nivel tienden a desperdiciar tiempo y espacio, y pueden complicar la especificación fuera del alcance de la habilidad del yacc para tratar con él. Normalmente el analizador léxico reconocerá los nombres de mes y devolverá una indicación de que se ha visto uno; en este caso, el nombre-mes sería un token.

Los caracteres literales , tales como la coma, tienen que pasarse a través del analizador léxico y se consideran tokens.

Los ficheros de especificación son muy flexibles. Es relativamente fácil añadir al ejemplo anterior la regla

```
fecha : nombre-mes '/' dia '/' año;
```

permitiendo

```
7 / 4 / 1776
```

como un sinónimo de

```
julio 4, 1776
```

En la mayoría de los casos, esta nueva regla se podría deslizar en un sistema de trabajo con un mínimo de esfuerzo, y muy poco peligro de interrumpir el input existente.

El input que se está leyendo puede que no reúna las especificaciones. Estos errores de input se detectan tan pronto como teóricamente sea posible con una exploración de izquierda a derecha; por lo tanto, no sólo se reduce la posibilidad de leer y calcular con unos datos de input erróneos, sino que los datos erróneos se pueden localizar rápidamente. El manejo de errores, proporcionado como parte de las especificaciones de input, permite el volver a introducir de nuevo errores en los datos, o la continuación del proceso de input después de saltar sobre los datos erróneos.

En algunos casos, yacc falla al producir un parser cuando se le da un grupo de especificaciones. Por ejemplo, las especificaciones pueden ser contradictorias, o pueden requerir un mecanismo de reconocimiento más potente que el disponible para el yacc. Los primeros casos representan errores de diseño; los últimos casos pueden a menudo corregirse haciendo el analizador léxico más potente, o volviendo a escribir algunas de las reglas gramaticales. Aunque yacc no puede manejar todas las especificaciones posibles, su potencia se compara favorablemente con sistemas similares; lo que es más, las construcciones que son difíciles de manejar para el yacc son también frecuentemente difíciles de manejar par nosotros mismos. Algunos usuarios han informado que la disciplina de formular especificaciones yacc válidas para su input han revelado errores de concepción o diseño al principio del desarrollo del programa.

Las próximas secciones siguientes describen:

- La preparación de las reglas gramaticales.
- La preparación de las acciones proporcionadas por el usuario relacionadas con las reglas gramaticales.
- La preparación de analizadores léxicos.
- El funcionamiento del parser
- Varias razones por las que el yacc puede ser incapaz de producir un parser partiendo de una especificación, y qué hacer acerca de ello.
- Un simple mecanismo para manejar las precedencias de los operadores en las expresiones aritméticas.
- Detección y recuperación de errores.
- El entorno operativo y los dispositivos especiales de los parsers que yacc produce.

- Algunos consejos sobre lo que podría mejorar el estilo y eficacia de las especificaciones.

9.2 Especificaciones

Los nombres se refieren o a tokens o a símbolos no terminales. El yacc requiere que los nombres de tokens sean declarados como tales. Además, por razones que se explicarán más tarde, a menudo es deseable incluir el analizador léxico como parte del fichero de especificación. Puede ser muy útil incluir también otros programas. Por lo tanto, cada fichero de especificación consta de tres secciones: la de declaraciones, (gramática) la de reglas, y la de programas. Las secciones están separadas por marcas de signos de tanto por ciento dobles %%. (el signo de tanto por ciento (%) se usa normalmente en las especificaciones del yacc como un carácter de escape).

En otras palabras, un fichero de especificaciones completo resulta algo así como

```
declaraciones
%%
reglas
%%
programas
```

La sección de declaración puede estar vacía. Además, si se omite la sección de programas, la segunda marca %% se puede omitir también; por lo tanto, la especificación yacc legal más pequeña es

```
%%
reglas
```

Los espacios en blanco, los tabuladores, y los caracteres newline se ignoran, excepto que éstos pueden no aparecer en nombres o en símbolos reservados de varios caracteres. Los comentarios pueden aparecer en cualquier lugar donde un nombre sea legal; éstos están incluidos entre /* ... */, como en C.

La sección de reglas está formada de una o más reglas gramaticales. Una regla gramatical tiene la forma:

```
A : CUERPO ;
```

A representa un nombre no terminal, y CUERPO representa una secuencia de cero o más nombres y literales. Los dos puntos o el punto y coma son puntuaciones yacc.

Los nombres pueden tener longitud arbitraria, y pueden estar compuestos de letras, el punto **(.)**, el símbolo de subrayado **(_)** y dígitos no iniciales. Las letras mayúsculas y minúsculas son distintas. Los nombres que se usan en el cuerpo de una regla gramatical pueden representar tokens o símbolos no terminales.

Un literal consta de un carácter incluido entre comillas simples **(')**. Al igual que en C, el signo barra invertida **(\ )** es un carácter de escape dentro de literales, y se reconocen todos los escapes C. Por lo tanto

' \n '	Newline
' \r '	Retorno de carro
' \" '	Comillas simples
' \\ '	Barra invertida
' \t '	Tabulador
' \b '	Backspace
' \f '	Salto de página
' \xxx '	"xxx" en octal

Por un número de razones técnicas, el carácter ASCII NUL (' \0 ' o 0) no se deberá usar nunca en reglas gramaticales.

Si hay varias reglas gramaticales con el mismo término izquierdo, entonces se puede usar la barra vertical para evitar volver a escribir el término izquierdo. Además, el punto y coma al final de una regla se puede omitir delante de una barra vertical. Por lo tanto las reglas gramaticales

```
A : B C D ;
A : E F ;
A : G ;
```

se pueden dar en yacc como

```
A : B C D
```

```
    | E F
    | G
    ;
```

No es necesario que todas las reglas gramaticales con el mismo término izquierdo aparezcan juntas en la sección de reglas gramaticales, aunque ésto hace el input mucho más legible, y es más fácil de cambiar.

Si un símbolo no terminal coincide con el literal vacío. Esto se puede indicar de una forma obvia:

```
vacio :  ;
```

Los nombres que representan tokens tienen que ser declarados; ésto se hace más fácilmente escribiendo

```
%token  nombre1  nombre2  ...
```

En la sección de declaraciones. (Ver las Secciones 3, 5, y 6 para una explicación más amplia). Cada símbolo no terminal tiene que aparecer en el término izquierdo de al menos una regla.

De todos los símbolos no terminal, uno, llamado el símbolo de comienzo, tiene una particular importancia. El parser está diseñado para reconocer el símbolo de comienzo; por lo tanto, este símbolo representa la estructura mayor y más general descrita por las reglas gramaticales. Por defecto, el símbolo de comienzo se toma como el término izquierdo de la primera regla gramatical de la sección de reglas. Es posible, y de hecho deseable, declarar explícitamente el símbolo de comienzo en la sección de declaraciones usando la palabra reservada %start:

```
% start  symbol
```

El final del input para el parser se señala por medio de un token especial llamado "señal de final". Si los tokens, hasta, pero no incluyendo, la señal de final forman una estructura la cual coincide con el símbolo de comienzo, la función parser regresa a quien la ha llamado después de que se reconoce la señal de final y éste acepta el input. Si la señal de final se ve en cualquier otro contexto, ésto es un error.

Es misión del analizador léxico proporcionado por el usuario devolver la señal de final cuando sea apropiado; ver la sección 3, a continuación. Normalmente la señal de final representa un estado de I/O obvio, tal como el final del fichero o el final de un registro.

9.3. Acciones

Con cada regla gramatical, el usuario puede asociar acciones que se van a llevar a cabo cada vez que se reconoce la regla en el proceso de input. Estas acciones pueden devolver valores, y pueden obtener los valores devueltos por las acciones anteriores. Lo que es más, el analizador léxico puede devolver valores por tokens, si es que ésto se desea.

Una acción es una sentencia C arbitraria, y como tal puede producir input y output, llamar subprograma, y alterar vectores externos y variables. Una acción se especifica por medio de una o más sentencias, incluidas entre llaves. Por ejemplo

```
A   :    ' ( ' B ' ) '
         {hello (l,"abc"); }
```

y

```
XXX  : YYY   ZZZ
         {printf ("a message\n");
          flag = 25; }
```

son reglas gramaticales con acciones

Para facilitar las comunicaciones entre las acciones y el parser, las sentencias de acciones se alteran ligeramente. El signo del dolar **($)** se usa como una señal para el yacc en su contexto.

Para devolver un valor, la acción asigna normalmente una pseudo-variable $$ a algún valor. Por ejemplo, una acción que lo único que hace es devolver el valor 1 es

```
{ $$ = 1;}
```

Para obtener los valores devueltos por acciones previas y por el analizador de léxico, la acción puede usar las pseudo-variables $1, $2, ... las cuales se refieren a los

valores devueltos por los componentes del término derecho de una regla, leyendo de izquierda a derecha. Por lo tanto, si la regla es

```
A : B C D ;
```

por ejemplo, entonces $2 tiene el valor devuelto por C, y $3 el valor devuelto por D.

Como ejemplo más concreto, considerese la regla

```
espr : ‘ ( ‘ expre ’ ) ’ ;
```

El valor devuelto por esta regla es normalmente el valor de la expr entre paréntesis. Esto se puede indicar por medio de

```
espr : ‘ ( ‘ expr ’ ) ’  { $$ = $2 ;}
```

Por defecto, el valor de una regla es el valor del primer elemento de ella ($1). Por lo tanto las reglas gramaticales de formato

```
A : B ;
```

frecuentemente no necesitan contener una acción explícita.

En el ejemplo anterior, todas las acciones vienen al final de sus reglas. A veces, es deseable obtener el control antes de que se haya reconocido completamente la regla. El yacc permite que una acción, sea escrita en medio de una regla lo mismo que al final. Esta regla se asume que devuelve un valor, accesible a través del mecanismo normal por medio de las acciones a la derecha de él. Por el contrario, puede acceder a los valores devueltos por los símbolos a su izquierda. Por lo tanto, en la regla

```
A   :   B
        {$$ = 1;}
        C
        { x = $2 ; y = $3;}
        ;
```

El efecto es poner 1 en x, y el valor devuelto por C en Y.

Las acciones que no terminan una regla las maneja el yacc creando un nuevo nombre de símbolo no terminal, y una nueva regla que hace coincidir este nombre con el literal vacío. La acción interior es la acción iniciada reconociendo esta regla añadida. Yacc trata igualmente el ejemplo anterior como si se hubiera escrito:

```
$ACT   :  /*empty*/
          {SS = 1 ;}
A      :  B $ACT C
          { x = $2 ; y= $3;}
```

En muchas aplicaciones el output no lo hace directamente las acciones; en vez de ésto, una estructura de datos, tales como un árbol de reconocimiento, se construye en memoria, y las transformaciones se aplican a él antes de que se genere el output. Los árboles de reconocimiento son particularmente fáciles de construir, dadas las rutinas para construir y mantener la estructura de árbol deseada. Por ejemplo, suponga que hay una función C node, escrita de forma que la llamada

```
node ( L, n1 , n2)
```

crea un punto de unión con la etiqueta L, y los descendientes n1 y n2,, y devuelve el índice del punto de unión recién creado. Entonces el árbol de reconocimiento se puede crear proporcionando acciones tales como:

```
expr   : expr '+' expr
         { $$ = node ('+', $1, $3)}
```

en la especificación.

El usuario puede definir otras variables que van a ser usadas por las acciones. Las declaraciones y definiciones pueden aparecer en la sección de declaraciones, incluidas entre signos %{ y %}. Estas declaraciones y definiciones tienen un ámbito global, por lo tanto éstas son conocidas por las sentencias de acción y por el analizador léxico. Por ejemplo:

```
%{int variable = 0; %}
```

se podía poner en la sección de declaraciones, haciendo variable accesible a todas las acciones. El parser del yacc sólo usa nombres que comienzan por yy; el usuario deberá evitar tales nombres.

En estos ejemplos, todos los valores son enteros; En una sección posterior se encontrará una explicación de los valores de otros tipos.

9.4. Análisis léxico.

El usuario debe proporcionar un analizador léxico para leer el input y comunicar tokens (con valores, si se desea) al parser. El analizador léxico es una función de valores enteros llamada yylex. La función devuelve un entero, llamado el número token, representando el tipo de token que se ha leído. Si hay un valor relacionado con este token, éste deberá ser asignado a la variable externa yylval.

El parser y el analizador léxico deben concordar en estos números de token para que se pueda realizar la comunicación entre ellos. Los números los puede elegir el yacc, o los puede elegir el usuario. En cualquier caso, el mecanismo "#define" de C se usa para permitir al analizador léxico devolver estos números simbólicamente. Por ejemplo, suponga que el nombre de token DIGIT ha sido definido en la sección de declaraciones del fichero de especificaciones del yacc. La porción relevante del analizador léxico puede ser como:

```
yylex( ) {
   extern int yylval;
   int c;
   ...
   c = getchar ( );
   ...
   switch ( c ) {
      ...
   case ' 0 ' :
   case ' 1 ' :
   ...
   case ' 9 ' :
      yylval = c-'0';
      return (DIGIT);
       ...
```

```
    }
...
```

La intención es devolver un número de token de DIGIT, y un valor igual al valor numérico del dígito. Siempre que el código del analizador léxico sea puesto en la sección de programas del fichero de especificaciones, el identificador DIGIT será definido como el número de token relacionado con el token DIGIT.

Este mecanismo lleva a producir analizadores léxicos claros y fácilmente modificables; el único inconveniente es la necesidad de evitar usar nombres de token en la gramática que sean reservados o significativos para el C o para el parser; por ejemplo, el uso de los nombres de token "if" o "while", es casi seguro que producirán dificultades importantes cuando se compile el analizador léxico. El nombre de token "error" está reservado para el manejo de errores, y no se debería usar libremente.

Como se ha mencionado anteriormente, los números de token pueden ser elegidos por el yacc o por el usuario. En la situación por defecto, los números los elige el yacc. El número de token por defecto de un carácter literal es el valor numérico del carácter en el conjunto de caracteres locales. Otros nombres tienen asignados números de token que comienzan en 257.

Para asignar un número de token a un token (incluyendo literales), la primera aparición del nombre de token o literal de la sección de declaraciones puede ir seguida inmediatamente de un número entero que no sea negativo. Este número entero se coge como el número de token del nombre o literal. Los nombres y literales no definidos por este mecanismo retienen sus definiciones por defecto. Es importante que todos los números de token sean distintos.

Por razones históricas, la marca de final debe tener el número de token 0 o ser negativa. Este número de token no lo puede redefinir el usuario. Por lo tanto, todos los analizadores léxicos deberán estar preparados para devolver un cero o un número negativo como número de token al llegar al final de su input.

Una herramienta muy útil para construir analizadores léxicos es el lex, explicado en la sección anterior. Estos analizadores léxicos están diseñados para funcionar en estrecha armonía con los parsers yacc. Las especificaciones de estos analizadores léxicos usan expresiones regulares en vez de reglas gramaticales. El lex se puede usar fácilmente para producir analizadores léxicos poco complicados, pero ahí quedan algunos lenguajes (tales como FORTRAN) los cuales no encajan en ningún encaje teórico, y cuyos analizadores léxicos se pueden crear a mano.

9.5. Cómo funciona el parser

El yacc convierte el fichero de especificaciones en un programa C, el cual reconoce el input de acuerdo con la especificación que se le ha dado. El algoritmo que se usa para ir desde la especificación hasta el parser es muy complicada, y no se explicará aquí (para más información, ver las referencias). El propio parser, sin embargo, es relativamente sencillo y simple, y entender cómo funciona, aún no siendo estrictamente necesario, hará que el tratamiento de recuperaciones de errores y de ambigüedades sea bastante más compresible.

El parser que produce yacc consta de una máquina de estado finito con un stack. El parser es también capaz de leer y recordar el siguiente token de input (llamado el lookahead token). El estado actual es siempre el que está en lo alto del stack .

A los estados de la máquina de estado finito se les da etiquetas de enteros pequeños; inicialmente, la máquina está en el estado 0, el stack contiene sólo el estado 0, y no se ha leído el lookahead token.

La máquina sólo tiene cuatro acciones disponibles para ellos, llamadas shitf, reduce, accept y error. Un movimiento del parser se hace de la forma siguiente:

1 Basado en su estado actual, el parser decide si necesita un lookahead para decidir qué acción se deberá llevara a cabo; si necesita una y no la tiene, llama al yylex para obtener el siguiente token.

2 Utilizando el estado actual, y el lookahead token si es necesario, el parser decide sobre su próxima acción, y la ejecuta. Esto puede hacer que algunos estados sean empujados dentro del stack o expulsados de él, y que el lookahead token sea procesado y que se deje solo.

La acción shift es la acción más común que toma el parser. Siempre que se toma

una acción shift, hay siempre un lookahead token. Por ejemplo, en el estado 56 puede haber una acción:

```
IF shift 34
```

lo cual quiere decir, en el estado 56, si el lookahead token es IF, el estado actual (56) se empuja hacia abajo en el stack, y el estado 34 se convierte en el estado en curso (en lo alto del stack). El lookahead token se borra.

La acción reduce evita que el stack crezca demasiado y se salga de sus límites. Las acciones reduce son apropiadas cuando el parser ha visto el lado derecho de una regla gramatical, y está preparado para anunciar que ha visto una parte de la regla, cambiando el lado derecho por el lado izquierdo. Puede ser necesario consultar el lookahead token para decidir si reducir, pero normalmente no se hace; de hecho, la acción por defecto (representada por un **.**) es a menudo una acción reduce.

Las acciones reduce están asociadas con reglas gramaticales individuales. A las reglas gramaticales también se les da números enteros pequeños, produciendo algo de confusión. La acción:

```
.  reduce 18
```

se refiere a la regla gramatical 18, mientras que la acción

```
IF shift 34
```

se refiere al estado 34.

Suponga que la regla que se está reduciendo es

```
A : x y z ;
```

La acción reduce depende del símbolo de la izquierda (A en este caso), y el número de símbolos del lado derecho (tres en este caso). Para reducir, primero saca los tres estados superiores del stack (en general, el número de estados que se han sacado es igual al número de símbolos del lado derecho de la regla). En efecto, estos estados son los que se han puesto en el stack mientras se reconocen x, y, y z, y que no servirán ya para ningún propósito útil. Después de sacar estos estados, un estado está sin cubrir, el cual

fue el estado en que estaba el parser antes de comenzar a procesar la regla. Usando este estado encubierto, y el símbolo en el lado izquierdo de la regla, lleva a cabo lo que está en efecto un shift de A. Se obtiene un nuevo estado, se empuja dentro del stack, y el parser continúa, Hay deferencias significativas entre el proceso del símbolo de la izquierda y un shift ordinario de un token, por lo tanto esta acción se llama una acción goto. En particular, el lookahead token es borrado por el shift, y no es afectado por un goto. En cualquier caso, el estado encubierto contiene una entrada tal como:

A goto 20

haciendo que el estado 20 sea empujado dentro del stack y se convierta en el estado en curso.

En efecto, la acción reduce expulsar los estados del stack para regresar al estado donde fue visto por primera vez en el lado derecho de la regla. El parser se comporta entonces como si el lado derecho de la regla estuviese vacío, no se expulsan estados del stack; el estado encubierto es, en efecto, el estado en curso.

La acción reduce es también importante en el tratamiento de acciones y valores proporcionados por el usuario. Cuando se reduce una regla, el código proporcionado con la regla se ejecuta antes de que el stack esté ajustado. Además de los estados del stack, otro stack, funcionando en paralelo con él, mantiene los valores devueltos del analizador léxico y las acciones. Cuando se produce un shift, la variable externa yylval se copia en el stack. Después del regreso del código del usuario, se efectúa la reducción. Cuando se ha terminado la acción goto, la variable externa yylval se copia en el stack. Las pseudo-variables $1, $2, etc, ... se refieren al stack.

Las otras dos acciones parser son conceptualmente mucho más sencillas. La acción accept indica que el input completo ha sido visto y que coincide con la especificación. Esta acción aparece sólo cuando el lookahead token es la señal de final, e indica que el parser ha ejecutado correctamente su función. La acción error, por otra parte, representa un lugar donde el parser no puede continuar nunca más su acción de acuerdo con la especificación. Los tokens de input que ha visto, junto con el lookahead token no puede ser seguido por nada que pudiese producir un input legal. El parser informa de un error, e intenta recobrarse de él (al contrario del error de detección) y continúa su trabajo; la recuperación de errores se explicará en la sección posterior.

Considerese el ejemplo siguiente:

```
%token DING DONG DELL
%%
rhyme  :  sound  place
       ;
sound  :  DING DONG
       ;
place  :  DELL
       ;
```

Cuando se llama a yacc con la opción -y, se produce un fichero llamado y.output, con una descripción del parser legible. El fichero y.output correspondiente a la gramática anterior (a la que se le han quitado algunas estadísticas del final) es:

```
state 0
  $accept : _rhyme $end

  DING shift 3
  . error

  rhyme goto 1
  sound goto 2

state 1
  $accept : rhyme_$end

  $end accept
  . error




state 2
  rhyme : sound_place

  DELL shift 5
  . error
```

```
    place goto 4

state 3
    sound : DING_DONG

    DONG shift 6
    .  error

state 4
    rhyme : sound place _ (1)

    .  reduce 1

state 5
    place : DELL_ (3)

    .  reduce 1

state 6
    sound : DING DONG_ (2)

    .  reduce 2
```

Nótese que, además de las acciones de cada estado, hay una descripción de las reglas del parser que se procesan en cada estado. El carácter de subrayado se usa para indicar qué es lo que se ha visto, y qué es lo que falta, en cada regla. Suponga que el input es

DING DONG DELL

Es instructivo seguir los pasos del parser mientras está procesando este input.

Inicialmente, el estado en curso es el estado 0. El parser necesita referirse al input para poder decidir entre las acciones disponibles en el estado 0, de forma que se lee el primer token, DING, convirtiendose en el lookahead token. La acción del estado 3 del token DONG es shift 6, por lo tanto el estado 6 se empuja dentro del stack, y se borra el lookahead. El stact contiene ahora 0, 3 y 6. En el estado 6, incluso sin consultar el lookahead, el parser reduce por la regla 2.

sound : DING DONG

Esta regla tiene dos símbolos en el lado derecho, por lo tanto se sacan del stack dos estados, el 6 y el 3. Consultando la descripción del estado 0, buscando un goto en sound se obtiene

sound goto 2

por lo tanto el estado 2 se empuja dentro del stack convirtiendose en el estado en curso.

En el estado 2, es necesario leer el siguiente token, DELL. La acción es shift 5, por lo tanto el estado 5 se empuja dentro del stack, el cual ahora tiene 0, 2, y 5 en él, y se borra el lookahead token. En el estado 5, la única acción es reducir por la regla 3. Esta tiene un símbolo en el lado derecho, por lo tanto un estado, el 5, se expulsa del stack, y el estado 2 se descubre. El goto en el estado 2 en place, el lado izquierdo de la regla 3, es el estado 4. Ahora el stack contiene 0, 2y 4. En el estado 4, la única acción es reducir por la regla 1. Hay dos símbolos en la derecha, por lo tanto los dos estados superiores se sacan, descubriendo de nuevo el estado 0. En el estado 0 hay un goto en rhyme haciendo que el parser introduzca el estado 1. En el estado 1, se lee el input; se obtiene la marca de final, indicada por $end en el fichero y.output. La acción del estado 1 cuando se ve la marca del final es aceptar, terminando correctamente el parse.

Se pide al lector que considere cómo funciona el parser cuando se confronta con un literal incorrecto tal como DING DONG DONG, DING DONG, DING DONG DELL DELL, etc. El pasar unos cuantos minutos con ésto y otros ejemplos sencillos tendrá probablemente su recompensa cuando surjan problemas en contextos más complejos.

9.6 Ambigüedad y conflictividad

Un grupo de reglas gramaticales es ambiguo si hay algún literal de input que se puede estructurar de dos o más formas diferentes. Por ejemplo, la regla gramática

expr : expr '-' expr

es una forma natural de expresar el hecho de que una manera de formar una expresión aritmética es poner otras dos expresiones juntas con un signo menos entre ellas. Desafortunadamente, esta regla gramatical no especifica completamente la forma en que

todos los inputs complejos deberán ser estructurados. Por ejemplo, si el input es

```
expr - expr - expr
```

la regla permite que este input sea estructurado o bien como

```
( expr - expr ) - expr
```

o como

```
expr -( expr - expr)
```

(La primera se llama asociación izquierda, la segunda, asociación derecha).

Yacc detecta tales ambigüedades cuando está intentando crear el parser. Es instructivo considerar el problema que confronta el parser cuando se especifica un input tal como

```
expr - expr - expr
```

Cuando el parser ha leído el segundo expr, el input que ha visto:

```
expr - expr
```

coincide con el lado derecho de la regla gramatical anterior. El parser puede reducir el input aplicando esta regla; después de aplicar la regla; el input se reduce a expr (el lado izquierdo de la regla). El parser leerá entonces la parte final del input.

```
- expr
```

y reduce de nuevo. El efecto de ésto es tomar la interpretación asociativa izquierda.

Alternativamente, cuando el parser ha visto

```
expr - expr
```

podría diferir la aplicación inmediata de la regla, y continuar leyendo el input hasta que haya visto

```
expr - expr - expr
```

Entonces podría aplicar la regla a los tres símbolos de la derecha, reduciendolos a expr y dejando

```
expr - expr
```

Ahora la regla puede reducirse una vez más; el efecto es tomar la interpretación asociativa derecha. Así, habiendo leído

```
expr - expr
```

el parser puede hacer dos cosas legales, un shift o una reducción, y no tiene forma de decidir entre ellos. Esto se llama un conflicto shift/reduce. También puede suceder que el parser tenga una elección de dos reducciones legales; ésto se llama un conflicto reduce/reduce. Nótese que nunca hay conflictos shift/shift.

Cuando hay conflictos shift/reduce o reduce/shift, el yacc todavía produce un parser. Hace ésto seleccionando uno de los pasos válidos siempre que tenga una elección. Una regla que describe qué selección hacer en una determinada situación se llama una regla desambiguante.

El yacc llama por defecto a dos reglas desambiguantes:

1. En un conflicto shift/reduce, por defecto se hace el shift.
2. En un conflicto reduce/reduce, por defecto se reduce aplicando la regla gramatical anterior (en la secuencia de input).

La regla 1 implica que reducciones son diferidas siempre que hay una selección, en favor de los shifts. La regla 2 da al usuario un control relativamente crudo sobre el comportamiento del parser en esta situación, pero los conflictos reduce/reduce se deberían evitar siempre que sea posible.

Los conflictos que pueden surgir por errores en el input o en la lógica, o por causa de las reglas gramaticales, aún siendo consistente, requieren un parser más complejo que el que el yacc puede construir. El uso de acciones dentro de las reglas pueden también producir conflictos, si la acción tiene que hacerse antes de que el parser pueda estar seguro de que regla se está reconociendo. En estos casos, la aplicación de reglas desambiguantes

es inapropiada, y conduce a un parser incorrecto. Por esta razón, el yacc siempre informa del número de conflictos shift/reduce y reduce/reduce resueltos por medio de la Regla 1 y de la Regla 2.

En general, siempre que es posible aplicar las reglas desambiguantes para producir un parser correcto, es posible escribir las reglas gramaticales de forma que se lean los mismos inputs, pero que no haya conflictos. Por esta razón la mayoría de los generadores parser previos han considerado conflictos como errores fatales. Nuestra experiencia ha sugerido que el volver a escribir ésto es de algún modo innatural, y produce parsers más lentos; por lo tanto, yacc producirá parsers incluso en presencia de conflictos.

Como un ejemplo de la potencia de las reglas desambiguantes, considere un fragmento de un lenguaje de programación que tiene una construcción if-then-else:

```
stat    : IF  '(' cond ')'  stat
        | IF  '(' cond ')'  stat  ELSE  stat
        ;
```

En estas reglas, IF y ELSE son tokens, cond es un símbolo no terminal describiendo expresiones condicionales (lógicas), y stat es un símbolo no terminal que describe sentencias. La primera regla será llamada la regla simple if, y la segunda, la regla if-else.

Estas dos reglas forman una construcción ambigua, puesto que el input de la forma

```
IF  ( C1 )  IF  ( C2 )  S1  ELSE  S2
```

se puede construir de acuerdo con estas reglas de dos formas:

```
IF  ( C1 ) {
          IF  ( C2 )  S1
          }
ELSE  S2
```

o

```
IF  ( C1 ) {
        IF  ( C2 )  S1
        ELSE  S2
      }
```

La segunda interpretación es la que se da en la mayoría de los lenguajes de programación que tienen esta construcción. Cada ELSE está asociado con el último IF inmediatamente delante del ELSE. En este ejemplo, considere la situación donde el parser ha visto

```
IF  ( C1 )  IF  ( C2 )  S1
```

y está mirando al ELSE. Este se puede reducir inmediatamente por la regla simple-if para obtener

```
IF  ( C1 )  stat
```

y después leer el input restante

```
ELSE  S2
```

y reduce

```
IF  ( C1 )  stat  ELSE  S2
```

por medio de la regla if-else. Esto lleva al primero de los agrupamientos anteriores del input.

Por otro lado, el ELSE se puede desplazar, leer S2, y después la porción derecha de

```
IF  ( C1 )  IF  ( C2 )  S1  ELSE  S2
```

se puede reducir por medio de la regla if-else para obtener

```
IF  ( C1 )  stat
```

lo cual se puede reducir por medio de la regla simple-if. Esta lleva al segundo de los

agrupamientos anteriores del input, lo cual es normalmente lo que se desea.

Una vez más el parser puede hacer dos cosas válidas - hay un conflicto shift/reduce. La aplicación de la regla desambiguante indica al parser seleccionar el shift en este caso, lo cual lleva al agrupamiento deseado.

Este conflicto shift/reduce surge sólo cuando hay un símbolo de input en curso particular, ELSE, y se han visto ya input particulares, tales como

IF (C1) IF (C2) S1

En general, puede haber muchos conflictos, y cada uno estará asociado con un símbolo del input y un grupo de inputs leídos previamente, los inputs leídos previamente están caracterizados por el estado del parser.

Los mensajes de conflictividad del yacc se entienden mejor examinando el fichero de output con la opción -v. Por ejemplo, el output correspondiente al estado de conflicto anterior podría ser:

```
23:     shift/reduce conflict\
        (shift 45, reduce 18) on ELSE
state 23

        stat  :  IF  ( cond )  stat_  (18)
        stat  :  IF  ( cond )  stat_ELSE  stat

ELSE shift 45
. reduce 18
```

La primera línea describe el conflicto, dando el estado y el símbolo de input. A continuación va la descripción de estado ordinaria, especificando las reglas gramaticales activas en el estado, y las acciones del parser. Recuerdese que el signo desubrayado marca

la porción de las reglas gramaticales que se han visto. Por lo tanto en el ejemplo, en el estado 23 el parser ha visto el input correspondiente a

```
IF  ( C1 )  stat
```

y las dos reglas gramaticales están activas en ese momento. El parser puede hacer dos cosas posibles. Si el símbolo de input es ELSE, es posible pasar al estado 45. El estado 45 tendrá, como parte de su descripción, la línea

```
stat  : IF  ( cond )  stat  ELSE_stat
```

puesto que ELSE habrá sido desplazada en este estado. De nuevo en el estado 23, la acción alternativa, descrita por medio de "**.**", se va a efectuar si el símbolo de input no se menciona explícitamente en las acciones anteriores; por lo tanto, en este caso, si el símbolo de input no es ELSE, el parser reduce por medio de la regla gramatical 18:

```
stat  : IF  '(' cond ')'  stat
```

Una vez más, téngase en cuenta que los números que van a continuación de los comandos shift se refieren a otros estados, mientras que los números que van a continuación de los comandos reduce se refieren a números de reglas gramaticales. En el fichero y.output, los números de reglas se imprimen después de estas reglas las cuales se pueden reducir. En la mayoría de los estados uno, habrá al menos una acción reduce posible en el estado, y ésto será el comando por defecto. El usuario que encuentra conflictos inesperados shift/reduce probablemente querrá mirar al output para decidir si las acciones por defecto son apropiadas. En casos realmente difíciles, el usuario puede necesitar saber más acerca del comportamiento y construcción del parser que lo que puede explicar aquí. En este caso, se podría consultar una de las referencias teóricas; los servicios de un guru local también pueden ser apropiados.

9.7 Precedencia

Hay una situación común donde las reglas dadas anteriormente para resolver conflictos no son suficientes; ésto es en el reconocimiento de expresiones aritméticas. La mayoría de las construcciones comúnmente usadas para expresiones aritméticas se pueden describir por la noción de niveles de precedencia para los operadores, junto con la información sobre asociatividad izquierda o derecha. Resulta que las gramáticas ambiguas con las reglas desambiguantes apropiadas se pueden usar para crear parsers que son más

rápidos y fáciles de escribir que los parsers construidos partiendo de gramáticas no ambiguas. La idea básica es escribir reglas gramaticales de la forma

```
expr : expr OP expr
```

y

```
expr : UNARY expr
```

para todos los operadores binarios y unarios deseados. Esto crea una gramática muy ambigua, con muchos conflictos. Como reglas desambiguantes, el usuario especifica la prioridad, de todos los operadores, u la asociatividad de los operadores binarios. Esta información es suficiente para permitir al yacc resolver los conflictos del parser de acuerdo con estas reglas, y construir un parser que acepte las precedencias y asociatividad deseadas.

La precedencia y asociatividad están unidas a tokens en la sección de declaraciones. Esto se hace por medio de una serie de líneas que comienzan con la palabra reservada del yacc: **%left**, **%rigth**, o **%nonassoc**, seguido de una lista de tokens. Todos los tokens de una misma línea se supone que tienen el mimo nivel de precedencia y asociatividad; las líneas están listadas en reglas de precedencia creciente. Por lo tanto,

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

describe la precedencia y asociatividad de los cuatro operadores aritméticos. Los signos más y menos son asociativos a la izquierda, y tienen menor precedencia que el asterisco y que la barra, los cuales son también asociativos a la izquierda. La palabra reservada %right se usa para describir operadores asociativos a la derecha, y la palabra reservada %nonassoc se usa para describir operadores, como el operador .LT. En el lenguaje FORTRAN, que no puede asociarse con ellos mismos; por lo tanto,

```
A .LT. B .LT. C
```

es ilegal en FORTRAN, y tal operador será descrito con la palabra reservada %nonassoc del yacc. Como un ejemplo del comportamiento de estas declaraciones, la descripción

```
%right '='
%left '+' '-'
%left '*' '/'
```

```
%%

expr    : expr '=' expr
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | NAME
        ;
```

se podría usar para estructurar el input

```
a = b = c*d - e - f*g
```

como sigue:

```
a = ( b = ( ( (c*d) - e)  - (f*g) ) )
```

Cuando se usa este mecanismo, los operadores unitarios deben, en general recibir una precedencia. A veces un operador unitario y un operador binario tienen la misma representación simbólica, pero precedencias diferentes. Un ejemplo es unitario y binario '-'; el signo menos unitario puede recibir la misma prioridad que la multiplicación, o incluso más alta, mientras que el signo menos binario tiene una prioridad menor que la multiplicación. La palabra reservada **%prec**, cambia el nivel de prioridad relacionado con una regla gramatical determinada. El %prec aparece inmediatamente después del cuerpo de la regla gramatical, antes de la acción o del punto y coma de cierre, y va seguido de un nombre de token o de literal. Esto produce que la precedencia de la regla gramatical sea la del nombre de token o literal siguiente. Por ejemplo, para hacer que el signo menos unitario tenga la misma prioridad que la multiplicación, las reglas deberán ser:

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

%%
```

```
expr    : expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | '-' expr %prec '*'
        | NAME
        ;
```

Un token declarado por medio de %left, %right, y %nonassoc no necesitan ser, pero pueden ser, declarados también por medio de %token.

Las precedencias y asociatividades las usa el yacc para resolver conflictos del parser; éstos dan cabida a las reglas desambiguantes. Formalmente, las reglas funcionan de la siguiente forma:

1. La precedencia y asociatividad se registra para esos tokens y literales que los tienen.

2. Una precedencia y asociatividad está asociada con cada regla gramatical; es la precedencia y asociatividad del último token o literal del cuerpo de la regla. Si se usa la construcción %prec, ésta contrarresta esta regla estándar. Algunas reglas gramaticales pueden no tener precedencias o asociatividad relacionadas con ellas.

3. Cuando hay un conflicto reduce/reduce, o hay un conflicto shift/reduce o bien el símbolo de input o la regla gramatical no tiene precedencia y asociatividad, entonces se usan las dos reglas desambiguantes especificadas al principio de la sección y se informa de los conflictos.

4. Si hay un conflicto shift/reduce, y ambas reglas gramaticales y el carácter de input tienen prioridad y asociatividad relacionada con ellos, entonces el conflicto se resuelve en favor de la acción (shift o reduce) relacionada con la prioridad más alta. Si las precedencias son las mismas, entonces se usa la asociatividad; la asociatividad izquierda implica reduce, la asociatividad derecha implica shift, y la no asociatividad implica error.

Los conflictos resueltos por precedencia no se cuentan en el número de conflictos de shift/reduce y reduce/reduce que informa el yacc. Esto quiere decir que los errores en la especificación de precedencias puede ocultar errores en la gramática de input; es una

buena idea ahorrar precedencias, y usarlas esencialmente a modo de libro de cocina, hasta que se haya obtenido alguna experiencia. El fichero y.output es muy útil para decir si el parser está realmente haciendo lo que se suponía tenía que hacer.

9.8 Manejo de errores.

El manejo de errores es un área extremamente difícil, y muchos de los problemas son de tipo semántico. Cuando se encuentra un error, por ejemplo puede ser necesario reclamar almacenamiento en árbol del parser, borrar o alterar entradas de tablas de símbolos, y, típicamente, poner switches para evitar generar más output.

Rara vez es aceptable parar todo el proceso cuando se encuentra un error. Es más útil continuar explorando el input para localizar más errores sintácticos. Esto lleva al problema de tener que volver a iniciar el parser después de un error. Una clase general de algoritmos para llevar a cabo ésto implica desechar un grupo de tokens del literal de input, e intentar ajustar el parser de forma que puede continuar el input.

Para permitir al usuario algún control sobre este proceso, el yacc proporciona un dispositivo simple, pero razonablemente general. El nombre de token **error** está reservado para el manejo de errores. Este nombre se puede usar en reglas gramaticales; en efecto éste sugiere lugares donde se esperan los errores, y la recuperación puede tener lugar. El parser levanta su stack hasta que entra en un estado donde el token error es legal. Entonces se comporta como si el token error fuese el token lookahead en curso, y lleva a cabo la acción que se ha encontrado. El token lookahead se resetea entonces al token que ha producido el error. Si no se han especificado reglas de error especiales, el proceso se para cuando se detecta un error.

Para evitar una cascada de mensajes de error, el parser, después de detectar un error, continúa en estado error hasta que se han leído e intercambiado con éxito tres tokens. Si se detecta un error cuando el parser está aún en estado de error. No se da ningún mensaje de error, y el token de input se borra silenciosamente.

Como un ejemplo, una regla de la forma

```
stat : error
```

significaría, en efecto, que en un error sintáctico el parser intentará saltarse la sentencia

en la cual se ha descubierto el error. Más precisamente, el parser explorará con anticipación buscando tres tokens que puedan legalmente seguir una sentencia y comenzar el proceso en el primero de éstos; si los principios de las sentencias no son suficientemente distintivos, puede crear un falso comienzo en medio de una sentencia y terminar informando de un segundo error donde de hecho no hay ningún error.

Se pueden usar acciones con estas reglas de error especiales. Estas acciones pueden intentar reinicializar tablas, reclamar espacio para la tabla de símbolo, etc.

Las reglas de error tales como la anterior son muy generales pero difíciles de controlar. Algo más fáciles son las reglas tales como

```
stat : error ';'
```

Aquí, cuando hay un error, el parser intenta saltar sobre la sentencia, pero lo hará saltando al siguiente ';'. Todos los tokens después del error y antes del siguiente ';' no se pueden intercambiar, y se desechan. Cuando se encuentra el ';' dicha regla se reducirá, y se llevará a cabo cualquier acción de limpieza relacionada con ella.

Otra forma de regla de error surge en aplicaciones interactivas donde puede ser deseable permitir que se vuelva a introducir una línea después de un error. Una regla de error posible podría ser

```
input : error '\n' { printf (" Reenter line: ");
        } input
      { $$ = $4; }
```

Hay una dificultad potencial con esta idea; el parser tiene que procesar correctamente tres tokens de input antes de que haya sido resincronizado correctamente después del error. Si la línea que se ha vuelto a introducir contiene un error en los primeros dos tokens, el parser borra los tokens que producen el error, y no genera ningún mensaje; ésto es claramente inaceptable. Por esta razón, hay un mecanismo que se puede usar para hacer que el parser crea que se ha recuperado totalmente de un error. La sentencia

yyerrok;

en una acción restablece el parser a su modo normal. El último ejemplo está mejor escrito

```
input   : error '\'
          { yyerrork;
            printf ("Reenter last line: " ) ;}
          input
          { $$ = $4; }
          ;
```

Como se ha mencionado anteriormente, el token visto inmediatamente después del símbolo de error es el token de input en el cual fue descubierto el error. A veces, ésto es inapropiado; por ejemplo, una acción de recuperación de error puede hacer propia la función de localizar el lugar correcto para continuar el input. En este caso, el token lookahead previo se debe borrar. La sentencia

yyclearin ;

en una acción tendrá este efecto. Por ejemplo, suponga que la acción después del error fuese llamar a alguna sofisticada rutina de resincronización, proporcionada por el usuario que ha intentado avanzar el input al principio de la siguiente sentencia válida. Después de llamar a esta rutina, el siguiente token devuelto por yylex será probablemente el primer token de una sentencia legal; el token ilegal antiguo tiene que ser desechado y el estado de error reseteado. Esto se podría hacer por medio de una regla como

```
stat : error
       { resynch ();
         yyerrork;
         yyclearin ; }
         ;
```

Estos mecanismos son realmente crudos, pero permiten una recuperación simple y bastante efectiva del parser de muchos errores. Además, el usuario puede obtener control para tratar con las acciones de errores requeridas por otras porciones del programa.

9.9 El entorno del yacc

Cuando el usuario introduce una especificación al yacc, el output es un fichero de programas C, llamado y.tab.c en la mayoría de los sistemas. La función producida por el

yacc se llama **yyparse** es una función de valores enteros. Cuando se la llama, ésta llama repetidamente al yylex, el analizador léxico proporcionado por el usuario para obtener tokens de input. Eventualmente, o bien se detecta un error, en cuyo caso (si no es posible recuperarse del error) yyparse devuelve el valor 1, o el analizador léxico devuelve el token de marca de final y el parser la acepta. En este caso, yyparse devuelve el valor 0.

El usuario debe proporcionar una cierta cantidad de entorno para este parser para poder obtener un programa. Por ejemplo, como con cada programa C, es necesario definir un programa llamado main, este eventualmente llama a yyparse. Además, una rutina llamada **yyerror** imprime un mensaje cuando se detecta un error sintáctico.

Estas dos rutinas tienen que ser proporcionadas de una forma u otra por el usuario. Para facilitar el esfuerzo inicial de usar yacc, ha sido proporcionada una librería con versiones por defecto de main y de yyerror. El nombre de esta librería depende del sistema; en muchos sistemas la librería se accede por medio de un argumento -ly al cargador. Para mostrar la trivialidad de estos programas por defecto, el fuente se da a continuación:

```
        main(){
                return ( yyparse ( ) );
                }
y
        #include <stdio.h>

        yyerror (s) char *s; {
                fprintf( stderr, "%s\n", s);
                }
```

El argumento de yyerror es un literal que contiene un mensaje de error, normalmente el literal " syntax error " (error de sintaxis). La aplicación normal deseará hacer algo mejor que esto. Normalmente, el programa deberá controlar el número de la línea de input, e imprimirlo junto con el mensaje cuando se detecta un error de sintaxis. La variable de enteros externa **yychar** contiene el número del token lookahead en el momento en que fue detectado el error; ésto puede ser de algún interés para producir mejores diagnósticos. Puesto que el programa main lo ha proporcionado probablemente el usuario (para leer argumentos, etc.) La librería yacc es útil sólo en pequeños proyectos,

o en las primeras etapas de los programas muy largos.

La variable entera externa **yydebug** tiene normalmente asignado el valor 0. Si tiene un valor distinto de cero, el parser imprimirá una descripción de sus acciones, incluyendo una discusión de cuáles son los símbolos de input que se han leído, y cuáles son las acciones del parser. Dependiendo del entorno operativo, puede ser posible asignar esta variable usando un sistema de depuración.

9.10 Preparación de especificaciones

Esta sección contiene consejos diversos sobre cómo preparar especificaciones eficientes, fáciles de cambiar y claras. Las subsecciones individuales son más o menos independientes

9.11 Estilo de input

Es difícil proporcionar reglas con acciones sustanciales y aún tener un fichero de especificaciones legibles.

1. Use letras mayúsculas para nombres de token, letras minúsculas para nombres no terminales. Esta regla le ayuda a conocer a quien culpar cuando las cosas van mal.

2. Ponga reglas gramaticales y acciones en líneas separadas. Esto permite cambiarlas sin una necesidad automática de cambiar la otra.

3. Ponga todas las reglas que sean del lado izquierdo juntas. Ponga las del lado izquierdo dentro sólo una vez, y deje que las reglas siguientes comiencen con una barra vertical.

4. Ponga un punto y coma únicamente después de la última regla con lado izquierdo, y ponga el punto y coma en una regla separada. Esto permite que se puedan añadir fácilmente nuevas reglas.

5. Indente cuerpos de reglas dos saltos de tabulador, y los cuerpos de acción tres saltos de tabulador.

Los ejemplos del texto de esta sección siguen este estilo (donde lo permite el espacio). El usuario debe decidirse sobre estas cuestiones estilísticas; el problema central,

de todos modos, es hacer las reglas visibles a través del código de acción.

9.12 Recursión izquierda.

El algoritmo usado por el parser del yacc anima a usar las llamadas reglas gramaticales recursivas a la izquierda: reglas de la forma

nombre: nombre resto_de_la_regla;

Estas reglas surgen frecuentemente cuando se escriben especificaciones de secuencias y listas:

```
list    : item
        | list ',' item
        ;
```

y

```
seq     : item
        | seq item
        ;
```

En cada uno de estos casos, la primera regla sólo será reducida para el primer item, y la segunda regla será reducida para el segundo y todos los items que van a continuación.

Con reglas recursivas a derecha, tales como

```
seq     : item
        | item seq
        ;
```

el parser será un poco más grande, y los items serán vistos reducidos, de derecha a izquierda. Más seriamente, un stack interno del parser estará en peligro de desbordamiento si se hubiese leído una secuencia muy larga. Por lo tanto, el usuario deberá usar la recursión de izquierda siempre que sea razonable.

Merece la pena considerar si una secuencia con cero elementos tiene algún significado, y si lo tiene, considere escribir la especificación de secuencias con una regla vacía:

```
seq     :  /* empty */
        | seq item
        ;
```

Una vez más, la primera regla será siempre reducida exactamente una vez, antes de que se haya leído el primer item y entonces la segunda regla será reducida una vez por cada item leído. Permitir secuencias vacías conduce a menudo a un aumento de generalidad. De todos modos, se pueden producir conflictos si se pide al yacc que decida qué secuencia vacía ha visto, cuando éste no ha visto suficiente para saber

9.13 Conexiones léxicas.

Algunas decisiones léxicas dependen del contexto. Por ejemplo, el analizador léxico puede normalmente querer borrar espacios en blanco, pero no dentro de literales entre comillas. O se puede introducir nombres dentro de una tabla de símbolos de las declaraciones, pero no de las expresiones.

Una forma de tratar esta situación es crear un flag global que el analizador léxico examina, y que las acciones asignan. Por ejemplo, suponga que un programa consta de 0 o más declaraciones seguidas de 0 o más sentencias. Considerese:

```
%{
int dflag;
%}
... otras declaraciones ...

%%
```

```
prog    : decls stats
        ;
```

```
decls   : /* empty */
            {dflag = 1;}
        |decls declaration
        ;
stats   : /* empty */
            dflag = 0;
        |stats statement
        ;

... otras reglas ...
```

El flag dflag es ahora 0 cuando se leen sentencias, y 1 cuando se leen declaraciones, excepto el primer token de la primera sentencia. Este token los tiene que ver el parser antes de que pueda decir que ha terminado la sección de declaración y que las sentencias han comenzado. En muchos casos, ésta única excepción de token no afecta la exploración léxica.

Este tipo de teoría de puerta trasera se puede sobrehacer. Sin embargo, representa una nueva forma de hacer algunas cosas que de otro modo serían difíciles.

9.14 Manejo de palabras reservadas

Algunos lenguajes de programación permiten al usuario utilizar palabras como if, la cual es reservada normalmente, como etiqueta o nombres de variables, siempre que tal uso no se enfrente con el uso legal de estos nombres del lenguaje de programación. Esto es terriblemente duro de hacer dentro de la estructura de yacc; es difícil pasar información al analizador léxico diciendole " esta forma de 'if' es una palabra reservada, y esta forma es una variable". El usuario puede hacer una tentativa, pero es difícil. Es mejor que las palabras sean reservadas; es decir, se prohíba usarlas como nombre de variables.

9.15 Simulación de error y de accept en acciones.

Las acciones del parser de error y accept se pueden simular en una acción usando las macros **YYACCEPT** y **YYERROR**. YYACCEPT hace que yyparse devuelva el valor 0; yyerror hace que el parser se comporte como si el símbolo de input en curso hubiese sido un error sintáctico; se llama a yyerror, y tiene lugar la recuperación del error. Estos mecanismos se usan para simular parsers con múltiples marcas de final o en comprobación de sintaxis de contexto sensitivo.

9.16 Acceder a valores en reglas incluidas

Una acción se puede referir a valores devueltos por acciones a la izquierda de la regla en curso. El mecanismo es simplemente el mismo que con acciones ordinarias, un signo de dolar seguido de un dígito, pero en este caso el dígito puede ser 0 o negativo. Considerese

```
sent :  adj  noun  verb  adj  noun
           {look at the sentence ...}
        ;
adj     : THE {$$  = THE;}
        |YOUNG { $$ = YOUNG;}
        ...
        ;
noun    : DOG {$$ = DOG;}
        | CRONE { if ($0 == YOUNG){
                        printf( "what? n" );
                        }
                  $$ = CRONE;
                 }
        ;
        ...
```

En la acción que va a continuación de la palabra CRONE, se hace una comprobación suponiendo que el token desplazado no fuese YOUNG. Obviamente, ésto sólo es posible cuando se sabe mucho sobre qué es lo que va a ir delante del símbolo noun del input. Pero de todos modos, a veces este mecanismo ahorrará un montón de problemas, especialmente cuando se van a excluir unas pocas combinaciones de una estructura regular.

9.17 Soportar tipos de valores arbitrarios

Por defecto, los valores devueltos por las acciones y el analizador léxico son números enteros. Yacc puede también soportar valores de otros tipos, incluyendo estructuras. Además, el yacc controla los tipos, e inserta los nombres de miembros de unión apropiados de forma que el parser resultante será estrictamente comprobado para verificar el tipo. El valor del stack del yacc está declarado como una unión de los diversos

tipos de valores deseados. El usuario declara la unión, y asocia nombres de miembros de unión a cada token y a cada símbolo no terminal que tiene un valor. Cuando el valor se referencia por medio de una construcción $$ o una construcción $n, el yacc insertará automáticamente el nombre de unión apropiado, de forma que no se producirán conversiones indeseadas. Además, los comandos de comprobación de tipo tales como lint(C) serán bastante más silenciosos.

Se usan tres mecanismos para proporcionar estos tipos. Primero, hay una forma de definir la unión; ésto lo debe hacer el usuario puesto que otros programas, notablemente el analizador léxico, deben saber acerca de los nombres de miembros de unión. Segundo, hay una forma de relacionar un nombre de miembro de unión con tokens y no terminales. Finalmente, hay un mecanismo para describir el tipo de esos pocos valores donde el yacc no puede determinar fácilmente el tipo.

Para declarar la unión, el usuario incluye en la sección de declaración:

```
%union{
        cuerpo de la unión...
        }
```

Esto declara el valor del stack de yacc, y las variables externas yylval y yyval, para tener un tipo igual a esta unión. Si el yacc ha sido llamado con la opción -d, la declaración de unión se copia en el fichero y.tab.h. Alternativamente, la unión puede ser declarada en un fichero de cabecera, y se usa un typedef para definir la variable YYSTYPE para que represente esta unión. Por lo tanto, el fichero de cabecera podía también haber dicho:

```
typedef union{
        cuerpo de la unión ...
        YYSTYPE;
        }
```

El fichero de cabecera tiene que estar incluido en la sección de declaraciones, usando el %{ y %}.

Una vez que YYSTYPE está definido, los nombres de los miembros de unión tienen que estar asociados con los diversos nombres de terminales y no terminales. La construcción

<nombre>

se usa para indicar un nombre de miembro de unión. Si ésto va a continuación de las palabras reservadas %token, %left, %right, y %nonassoc, el nombre de miembro de unión está asociado con los tokens listados. Por lo tanto, decir

```
%left <optype> '+' '-'
```

producirá que cualquier referencia a valores devueltos por estos dos tokens sea linkado con el nombre de miembro de unión optype. Otra palabra reservada, %type, se usa similarmente para asociar nombre de miembros de unión con no terminales. Por lo tanto, se puede decir

```
%type <nodetype> expr stat
```

Hay quedan un par de casos donde estos mecanismos son insuficientes. Si hay una acción dentro de una regla, el valor devuelto por esta acción no tiene un tipo predefinido. Similarmente, la referencia a valores de contexto izquierdo (tales como $0 - ver la subsección previa) deja al yacc sin una forma fácil de conocer el tipo. En este caso, se puede imponer un tipo en la referencia insertando un nombre de miembro de unión entre < y >, inmediatamente después del primer $. Un ejemplo de ésto es

```
rule    : aaa {$<intval>$ = 3;} bbb
            {fun( $<intval>2, $<other>0);}
        ;
```

Esta sintaxis tiene poco que recomendar, pero la situación surge muy raramente.

En una posterior sección se da un ejemplo de especificación. Los dispositivos de esta subsección no se activan hasta que se usan, en particular, el uso de %type activará esos mecanismos. Cuando se usan, hay un nivel de verificación bastante estricto. Por ejemplo, se crean diagnósticos al usar $n o $$ para referirse a algo que no tiene tipo definido Si estos dispositivos no se activan, se usa el valor del stack del yacc para contener valores enteros.

9.18 Una pequeña calculadora de escritorio

Este ejemplo da la especificación completa de yacc para una pequeña calculadora

de escritorio: La calculadora de escritorio tiene 26 registros, marcados a hasta z, y acepta expresiones aritméticas formadas de los operadores +, -, *, /, % (operador mod), & (and), | (or), y la asignación. Si una expresión a nivel superior es una asignación, el valor no se imprime; en caso contrario se imprime. Como en C, se asume que un entero que comienza por 0 (cero) es un número octal, en caso contrario, se asume que es decimal.

Como ejemplo de una especificación del yacc, la calculadora de escritorio hace el trabajo de mostrar como se usan las precedencias y las ambigüedades y demuestra una recuperación de error sencilla. Las mayores sobresimplificaciones son que la fase de análisis léxico es mucho más sencilla que para la mayoría de las aplicaciones, y se produce el output inmediatamente línea a línea. Nótese la forma en que se leen los enteros decimales y octales en las reglas gramaticales; este trabajo se hace probablemente mejor por medio del analizador léxico.

```
%{
#include <stdio.h>
#include <ctype.h>

int regs[26]
int base;

%}
%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* Precedencia para el signo menos unitario */

%%    /* Comienzo de la sección de reglas */
```

```
list    :    /* vacía */
        | list stat '\n'
        | list error '\n'
            yyerrok;
        ;
stat    : expr
          { printf("%d\n",$1); }
        | LETTER '=' expr
          { regs[$1] = $3;  }
        ;
expr    : '(' expr ')'
                { $$ = $2 ;}
        | expr '+' expr
                { $$ = $1 + $3; }
        | expr '-' expr
                { $$ = $1 - $3; }
        | expr '*' expr
                { $$ = $1 * $3; }
        | expr '/' expr
                { $$ = $1 / $3; }
        | expr '%' expr
                { $$ = $1 % $3; }
        | expr '&' expr
                { $$ = $1 & $3; }
        | expr '|' expr
                { $$ = $1 | $3; }
        | '-' expr %prec UMINUS
                { $$ = - $2; }
        | LETTER
                { $$ = regs [$1];}
        | number
        ;

number: DIGIT
                {$$ = $1; base = ( $1 == 0) ? 8 : 10;}
        | number DIGIT
                { $$ = base * $1 + $2;}
        ;
```

```
%%     /* comienzo de los programas */

yylex()          /* rutina de análisis léxico*/
        /* devuelve LETTER por un aletra minúscula */
        /* yylval = 0 hasta 25 */
        /* devuelve DIGIT por un dígito */
        /* yylval = 0 hasta 9 */
        /* todos los otros caractreres */
        /* se devuelven inmediatamente */
int c;

while ( (c=getchar()) == ' ')
        {/* salta espacios en blanco */}

/* c ahora no es un espacio en blanco */

if( islower ( c ) ) {
        yylval = c - 'a';
        return ( LETTER );
        }
if( isdigit ( c ) ) {
        yylval = c - '0';
        return ( DIGIT );
        }
return ( c )
}
```

9.19 Sintaxis del input de Yacc

Esta sección tiene una descripción de la sintaxis de input del yacc, como una especificación del yacc. Las dependencias de contexto, etc., no se tienen en cuenta. Irónicamente, el leguaje de especificación de input del yacc se especifica más naturalmente como una gramática LR(2); la parte difícil viene cuando se ve un identificador en una regla inmediatamente después de una acción. Si este identificador va seguido de un signo de dos puntos (:), es el comienzo de la regla siguiente; en caso contrario es una continuación de la regla en curso, lo que es simplemente tener una acción incluida en ella. Según se ha implementado el analizador léxico mira hacia adelante después de ver un identificador, y decide si el siguiente token (saltando los espacios en blanco, newlines, comentarios, etc.

) es un signo dos puntos, si lo es , devuelve el token C_IDENTIFIER. En caso contrario, devuelve IDENTIFIER. Los literales (cadenas de caracteres incluidas entre comillas) también se devuelven como IDENTIFIER, pero nunca momo parte de C_IDENTIFIER.

```
  /* Gramática para el input de Yacc*/
  /* entidades básicas */
%token IDENTIFIER
        /* incluye identificadores y literales */
%token C_IDENTIFIER
        /* identificador seguido de dos puntos */
%token NUMBER /* [0-9]+ */

  /* palabras reservadas : %type => TYPE,
        %left => LEFT, etc. */

%token LEFT RIGHT NONASSOC TOKE PREC \
        TYPE START UNION
%token MARK  /* the %% mark */
%token LCURL  /* the %{ mark */
%token RCURL  /* the %} mark */

  /*  los literales de caracteres ascii tienen su propio significado */
%start spec

%%

spec    : defs MARK rules tail
        ;
tail    : MARK { Eat up the rest of the file}
        |    /* vacío: la segunda MARK es opcional */
        ;
defs    ;    /* vacío*/
        | defs def
        ;
def     : START IDENTIFIER
        | UNION { copia la definición de unión en el output}
        | LCURL { Copia código C en el fichero de output} RCURL
        | ndefs rword tag nlist
```

```
        ;
rword   : TOKEN
        | LEFT
        | RIGHT
        | NONASSOC
        | TYPE
        ;
tag     :     /* vacío : la etiqueta de unión es opcional*/
        | '<' IDENTIFIER '>'
        ;
nlist   : nmno
        | nlist nmno
        | nlist ',' nmno
        ;
nmno    : IDENTIFIER                /* literal ilegal con %type*/
        | IDENTIFIER NUMBER         /* ilegal con %type */
        ;

  /* Sección de reglas */
rules   : C_IDENTIFIER rbody prec
        | rules rule
        ;
rule    : C_IDENTIFIER rbody prec
        | '|' rbody prec
        ;
rbody   :   /* vacío */
        | rbody IDENTIFIER
        | rbody act
        ;
act     : '{' (Copia acción, traduce $$) '}'
        ;
prec    :   /* vacío */
        | PREC IDENTIFIER
        | PREC IDENTIFIER act
        | prec ';'
        ;
```

9.20 Un ejemplo avanzado

Esta sección da un ejemplo de una gramática usando parte de los dispositivos avanzados discutidos en secciones anteriores. El ejemplo de la calculadora de escritorio se modifica para proporcionar una calculadora de escritorio que hace aritmética de intervalos de punto flotante. La calculadora entiende las constantes de punto flotante, las operaciones aritméticas +, -, *, /, y signo menos unitario -, y = (asignación), y tiene 26 variables de punto flotante, desde a hasta z. Además, también entiende intervalos, escritos

(x , y)

donde x es menor o igual a y. Hay 26 variables de valores de intervalos A hasta Z que también se pueden usar. Las asignaciones no devuelve ningún valor, y no imprimen nada, mientras que las expresiones imprimen el valor (flotante o intervalo).

Este ejemplo explora un gran número de dispositivos muy interesantes del yacc y del C. los intervalos están representados por una estructura, consistente en los valores de final del punto de la izquierda y de la derecha, almacenados como valores de precisión doble. A esta estructura se le dá un nombre de tipo, INTERVAL, usando typedef. El valor del stack de yacc también puede contener escalares de punto flotante, y enteros (usados para indexar dentro de matrices que contienen los valores de las variables). Nótese que esta estrategia completa depende grandemente de que sea posible asignar estructuras y uniones en C. En efecto, muchas de las acciones llaman a funciones que también devuelven estructuras.

También hay que señalar el uso de YYERROR para manejar condiciones de error: división de un intervalo que contiene 0, y un intervalo presentado en la regla erróneo. En efecto, el mecanismo del yacc de recuperación de un error se usa para desechar el resto de la línea que contiene el error.

Además de la mezcla de tipos del stack, esta gramática también demuestra un interesante uso de la sintaxis para controlar el tipo (escalar o intervalo) de expresiones intermedias. Nótese que un escalar se puede ascender automáticamente a un intervalo si el contexto demanda un valor intervalo. Esto produce un gran número de conflictos cuando se ejecuta la gramática a través de yacc: 18 Shift/Reduce y 26 Reduce/Reduce. El problema se puede ver mirando a las dos líneas de input

2.5 + (3.5 - 4.)

y

2.5 + (3.5 , 4.)

Téngase en cuenta que el 2.5 se va a usar en una expresión de valor intervalo en el segundo ejemplo, pero este hecho no se conoce hasta que se lee la coma (,); para entonces, 2.5 ha terminado, y el parser no puede volver a atrás y cambiar de idea. Más generalmente, podría ser necesario buscar un número arbitrario de tokens para decidir si convertir un escalar en un intervalo. Este problema se salva teniendo dos reglas para cada operador de valor intervalo binario; uno cuando el operando izquierdo es un escalar, y uno cuando el operando izquierdo es un intervalo. En este segundo caso, el operando de la derecha tiene que ser un intervalo, por lo tanto la conversión se aplicará automáticamente.

Pero de todos modos, hay todavía muchos casos donde la conversión puede o no puede ser aplicada, llegando a los conflictos anteriores. Estos se resuelven listando las reglas que producen escalares primero en el fichero de especificación; de esta forma, los conflictos se resolverán en la dirección de mantener expresiones de valores escalares como valores escalares hasta que sea necesario convertirlos en intervalos.

Esta forma de manejar diversos tipos es muy instructiva, pero no muy general. Si hubiera muchos tipos de expresiones, en vez de sólo dos, el número de reglas necesarias aumentaría dramáticamente, y los conflictos aun más dramáticamente. por lo tanto, mientras que este ejemplo es instructivo, es una práctica mejor en un entorno de lenguaje de programación el mantener el tipo de información como parte del valor, y no como parte de la gramática.

Finalmente, una palabra acerca de los analizadores léxicos. La única característica no usual es el tratamiento de constantes de punto flotante. la rutina de librería C atof se usa para hacer la conversión de un literal de caracteres a un valor de doble precisión. Si el analizador léxico detecta un error, éste responde devolviendo un token que es ilegal en la gramática, provocando un error sintáctico en el parser, y por supuesto la recuperación de errores.

```
%{

#include <stdio.h>
#include <ctype.h>

typedef struct interval {
        double lo, hi;
```

```
        } INTERVAL;

INTERVAL vmul ( ), vdiv ( );

double atof ();

double dreg[26];
INTERVAL verg[26];

%}

%start lines

%union {
        int ival;
        double dval;
        INTERVAL vval;
        }

%token <ival> DREG VREG
        /* indica dentro de dreg, matrices vreg */

%token <dval> CONST
        /* constante de punto flotante */

%type <vval> dexp
        /* expresión */

%type <dval> vexp
        /* expresión intervalo*/

  /* Información de precedencia de los operadores */

%left '+' '-'
%left '*' '/'
%left UMINUS   /* precedencia para el signo menos unitario */

%%
```

```
lines   : /* vacío */
        | lines line
        ;

line    : dexp '\n'
                {printf ( "%15.8f\n", $1);}
        | vexp  '\n'
                {printf ( "( %15.8f, %15.8f )\n", $1.lo, $1.hi);}
        | DREG '=' dexp '\n'
                { dreg[$1] = $3 ; }
        | VREG '=' vexp '\n'
                { vreg[$1] = $3 ; }
        | error '\n'
                {yyerrok;}
        ;

dexp    : CONST
        | DREG
                { $$ = dreg [$1];}
        | dexp '+' dexp
                { $$ = $1 + $3;}
        | dexp '-' dexp
                { $$ = $1 - $3;}
        | dexp '*' dexp
                { $$ = $1 * $3;}
        | dexp '/' dexp
                { $$ = $1 / $3;}
        | '-' dexp % prec UMINUS
                { $$ =  - $2;}
        | '(' dexp ')'
                { $$ =  $2;}
        ;

vexp    : dexp
                { $$.hi = $$.lo = $1;}
        | '(' dexp ',' dexp ')'
                {
                  $$.lo = $2;
```

```
                $$.hi = $4
                If ($$.lo > $$.hi) {
                        printf (“interval out order\n”);
                        YYERROR;
                        }
                }
        | VREG
                { $$ = vreg[$1] ;}
        |vexp ‘+’ vexp
                { $$.hi = $1.hi + $3.hi;
                  $$.lo = $1.lo + $3.lo;}
        |vexp ‘+’ vexp
                { $$.hi = $1 + $3.hi;
                  $$.lo = $1 + $3.lo;}
        |vexp ‘-’ vexp
                { $$.hi = $1.hi - $3.lo;
                  $$.lo = $1.lo - $3.hi;}
        |vexp ‘-’ vexp
                { $$.hi = $1 - $3.lo;
                  $$.lo = $1 - $3.hi;}
        |vexp ‘*’ vexp
                { $$ = vmul ( $1.lo, $1.hi, $3);}
        |vexp ‘*’ vexp
                { $$ = vmul ( $1, $1, $3);}

        |vexp ‘/’ vexp
                { if(dcheck ( $3 ) ) YYERROR;
                  $$ = vdiv ( $1.lo, $1.hi, $3);}
        |vexp ‘/’ vexp
                { if(dcheck ( $3 ) ) YYERROR;
                  $$ = vdiv ( $1, $1, $3);}
        | ‘-‘ vexp %prec UMINUS
                { $$.hi = .$2.lo; $$.lo = -$2.hi;}
        | ‘(‘ vexp ‘)’
                { $$ = $2;}
        ;

%%
```

```
#define BSZ 50
        /* tamaño del buffer para fp números  */
  /* análisis léxico */

yylex(){
        register c;
                /* salta los espacios en blanco */
        while ( (c = getchar( )) == ' ' )

        If ( isupper ( c ) ) {
                yylval,ival = c- 'A';
                return ( VREG);
                }
        If ( islower ( c ) ) {
                yylval,ival = c- 'a';
                return ( DREG);
                }



        If ( isdigit ( c ) || c== '.' ) {
                /* engulle dígitos, puntos, exponentes */

                char buf[BSZ+1], *cp = buf;
                int dot = 0, exp = 0;

                for( ;  (cp-buf)<BSZ ; ++cp, c=getchar( ) ){
                        *cp = c;
                        if ( isdigit ( c ) ) continue;
                        if ( c == '.' ) {
                                if (dot++ || exp) return ('.');
                                  /* lo anterior produce un error de sintaxis */
                                continue;
                                }
                        if ( c == 'e') {
                                if (exp++) return ('e');
                                        /* lo anterior causa error de sintaxis */
                                 continue;
```

```
                        }
                /* final del número*/
                break;
                }
        *cp = '\0';
        if ( (cp- buf) >= BSZ )
                printf("constante demasiado larga: truncada\n");
        else ungetc (c, stdin);
                /* lo anterior 'push' el último carácter que se ha leído*/
        yylval.dval = atof (buf);
        return ( CONST);
        }
return (c);
}

INTERVAL hilo ( a, b, c, d) double a, b, c, d;\
        /* devuelve el menor intervalo que contiene */
        /* a, b, c y d usados por *, / rutinas*/
        INTERVAL v;

        if( a>b) {v.hi = a; v.lo = b;}
        else {v.hi = b; v.lo = a;}



        if ( c>d) {
                if (c>v.hi)     v.hi = c;
                if (d<v.lo)     v.lo = d;
                }
        else    {
                if (d>v.hi)     v.hi = d;
                if (c<v.lo)     v.lo = c;
                }
        return (v);
}

INTERVAL vmul (a, b, v) double a, b,; \
        INTERVAL V; {
```

```
                return ( hilo (a*v.hi, a*v.lo, b*v.hi, b*v.lo));
                }

dcheck ( v) INTERVAL v ; {
        if (v.hi >= 0. && v.lo <= 0. ){
                printf ("divisor interval contains 0. \n);
                return (1);
                }
        return(0);
        }

INTERVAL vdiv (a, b, v )double a, b; \
        INTERVAL v ; {
                return (hilo (a/v.hi, a/v.lo, b/v.hi,b/v.lo));
```

9.21 Antiguas características

Esta sección menciona sinónimos y características las cuales se soportan por razón de continuidad histórica, pero, por diversas razones, no se recomiendan.

1. Los literales también se pueden delimitar por medio de comillas dobles (").

2. Los literales pueden tener más de un carácter de largo. Si tolos los caracteres son alfabéticos, numéricos, o signos de subrayado, el número de tipo del literal está definido justo como si el literal no tuviera alrededor de él signo de comillas. De otra forma, es difícil encontrar el valor para tales literales. El uso de literales multicarácter es probable que confunda a esos usuarios que o están familiarizados con el yacc, puesto que sugiere que el yacc esté haciendo un trabajo que tendría realmente que hacer el analizador léxico

3. La mayoría de los lugares donde '%' es legal, se puede usar la barra invertida (\). En particular, la doble barra invertida (\\) es lo mismo que %%, \left lo mismo que %left, etc.

4. Hay una gran cantidad de otros sinónimos:
   %< es lo mismo que %left
   %> es lo mismo que %right
   %binary y %2 son lo mismo que %nonassoc

%0 y %term son lo mismo que %token
%= es lo mismo que %prec

5. Las acciones pueden también tener la forma
   = { ... }
   y las llaves se pueden suprimir si la acción es una sola sentencia C.

6. El código C entre %{ y %} se solía permitir al principio de la sección de regla, así como en la sección de declaración.