PRÁCTICA OBLIGATORIA

PROCESADORES DE LENGUAJES

Fernando López Berrocal, Javier Romero García y Loreto Uzquiano Esteban.

ÍNDICE

DESCRIPCIÓN REGLAS LÉXICAS 2

COMMENT 2

PROCEDIMIENTO REALIZADO PARA TRANSFORMAR LA GRAMÁTICA A LL(1) 2

CABECERAS 3

SIGUIENTES 4

DIRECTORES 5

EXPLICACIÓN ERRORES CONTENIDOS EN LOS 4 CASOS DE PRUEBA 7

DESCRIPCIÓN IMPLEMENTACIÓN DE LA NOTIFICACIÓN DE ERRORES 10

DESCRIPCIÓN RESOLUCIÓN DE LA RECUPERACIÓN DE ERRORES 11

DESCRIPCIÓN REGLAS LÉXICAS

Las reglas léxicas más importantes son:

- **IDENT**. Ristra de símbolos que identifican a una variable o una función.
- **CONSTINT**. Constantes numéricas enteras.
- **CONSTFLOAT.** Constantes numéricas decimales.
- **CONSTLIT**. Constantes literales
- **COMMENT**. Comentarios de propósito general.

A continuación, damos una pequeña explicación de la regla léxica más importante (COMMENT).

COMMENT

```
COMMENT: '//' ~[\r\n]* ('\r'? '\n')?
```

Este es un comentario de línea. Empezará por '//'. Luego pueden existir símbolos que no sean saltos de línea ni retornos de carro (~[\r\n]*). Por último, acaba con un salto de línea (('\r'? '\n')?) que opcionalmente tiene el retorno de carro.

```
COMMENT: '/*' .*? '*/';
```

Este es un comentario en varias líneas. Puede existir cualquier símbolo (.*?) dentro de '/*' y '*/'.

PROCEDIMIENTO REALIZADO PARA TRANSFORMAR LA GRAMÁTICA A LL(1)

Antes de empezar a calcular los conjuntos directores de la gramática, fue necesario cambiar varias reglas, ya que presentaban recursiones por la izquierda o eran no deterministas.

Primero se arreglaron los casos en los que la gramática presentaba no determinismo.

El primer caso siendo las producciones que creaban listas de un elemento, con una producción que continuaba la lista, y otra que la acababa. Para arreglar esto cambiamos la producción cuyo consecuente acababa la lista a una producción vacía, y añadimos el elemento justo antes del antecesor en las demás producciones donde estuviese.

Las producciones modificadas por esto son "partes" y "program".

En el otro caso, las producciones que provocaban el no determinismo se podían agrupar en una nueva.

Las producciones modificadas por esto son "sent", "restpart" y "factor", para las cuales se crearon las producciones "ident1", "ident2" y se modificó "lexp".

Para arreglar las recursiones por la izquierda, creamos nuevas reglas y modificamos las reglas originales para que, entre las dos, se mantuviese la estructura original de las producciones eliminando la recursión.

Así, "listparam" se dividió en "listparam1" y "listparam2"; "lid" se dividió en "lid1" y "lid2"; "exp", en "exp1" y "exp2"; y, por último, "lcond" se dividió en "lcond1" y "lcond2".

CABECERAS

Los conjuntos cabeceras se muestran a continuación:

CAB(axioma) = CAB(program) = {'#define', 'void', 'int', 'float'}

CAB(program) = CAB(defines) ∪ CAB(part) = {'#define', 'void', 'int', 'float'}

CAB(defines) = {'#define', λ}

CAB(ctes) = {CONSTINT, CONSTFLOAT, CONSTLIT}

CAB(pautes) = CAB(paut) ∪ {λ} = {'void', 'int', 'float', λ}

CAB(paut) = CAB(type) = {'void', 'int', 'float'}

CAB(lcond1) = CAB(cond) ∪ {'!'} = {IDENT, '(', CONSTINT, CONSTFLOAT, CONSTLIT, '!'}

CAB(lcond2) = CAB(op1) ∪ {λ} = {'||', '&&', λ}

CAB(op1) = {'||', '&&'}

CAB(cond) = CAB(exp1) = {IDENT, '(', CONSTINT, CONSTFLOAT, CONSTLIT}

CAB(opr) = {'==', '<', '>', '>=', '<='}

CAB(restpaut) = {'('}

CAB(params) = CAB(lustparam1) ∪ {'void'} = {'void', 'int', 'float'}

CAB(lustparam1) = CAB(type) = {'void', 'int', 'float'}

CAB(lustparam2) = {λ, ','}

CAB(bloq) = {'{'}

CAB(type) = {'void', 'int', 'float'}

CAB(sentlust) = CAB(sent1) ∪ {λ} = {'void', 'int', 'float', IDENT, 'return', 'if', 'while', 'do', 'for', λ}

CAB(sent1) = CAB(type) ∪ CAB(ident1) ∪ {'return', 'if', 'while', 'do', 'for'} = {'void', 'int', 'float', IDENT, 'return', 'if', 'while', 'do', 'for'}

CAB(ident1) = {IDENT}

CAB(ident2) = {'=', '('}

CAB(lid1) = {IDENT}

CAB(lid2) = {',', λ}

CAB(lexp1) = CAB(exp1) ∪ {λ} = {IDENT, '(', CONSTINT, CONSTFLOAT, CONSTLIT, λ}

CAB(lexp2) = {',', λ}

CAB(exp1) = CAB(factor1) = {IDENT, '(', CONSTINT, CONSTFLOAT, CONSTLIT}

CAB(exp2) = CAB(op) ∪ {λ} = {'+', '-', '*', '/', '%', λ}

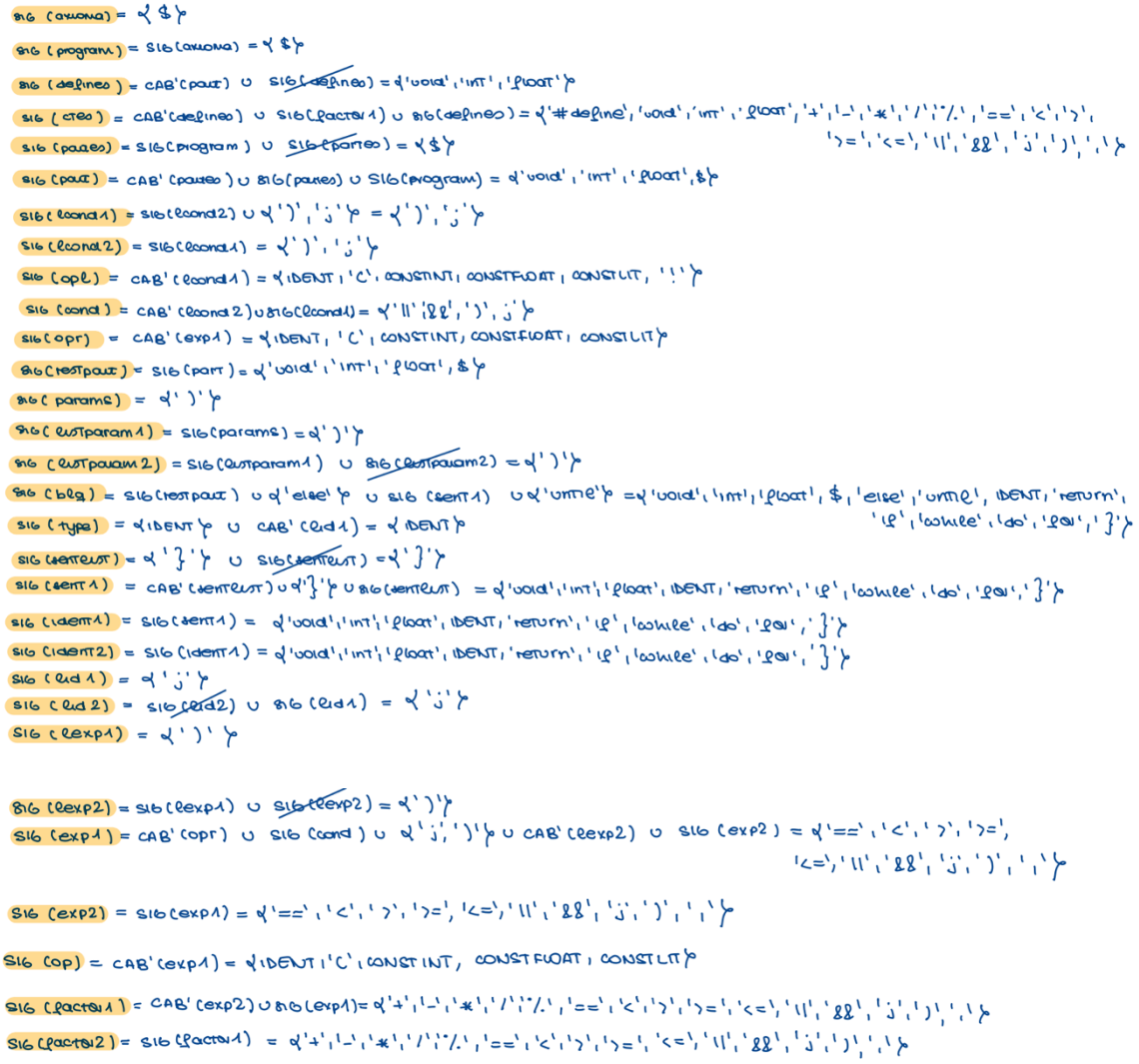
CAB(op) = {'+', '-', '*', '/', '%'}

CAB(factor1) = {IDENT, '('} ∪ CAB(ctes) = {IDENT, '(', CONSTINT, CONSTFLOAT, CONSTLIT}

CAB(factor2) = {'(', λ}

SIGUIENTES

Los conjuntos siguientes se muestran a continuación:



DIRECTORES

Los conjuntos directores se muestran a continuación:

DIR(axioma → program) = CAB'(program) = {'#define', 'void', 'int', 'float'}

DIR(program → defines part partes) = CAB'(defines) ∪ DIR(defines → part defines) = CAB'(defines) ∪ CAB'(part) = {'#define', 'void', 'int', 'float'}

DIR(defines → '#define' IDENT ctes defines) = {'#define'}

DIR(defines → λ) = SIG(defines) = {'void', 'int', 'float'}

DIR(ctes → CONSTINT) = {CONSTINT}

DIR(ctes → CONSTFLOAT) = {CONSTFLOAT}

DIR(ctes → CONSTLIT) = {CONSTLIT}

DIR(partes → part partes) = CAB'(part) = {'void', 'int', 'float'}

DIR(partes → λ) = SIG(partes) = {$}

DIR(part → type IDENT restpart) = CAB'(type) = {'void', 'int', 'float'}

DIR(cond1 → cond cond2) = CAB'(cond) = {IDENT, '(', CONSTINT, CONSTFLOAT, CONSTLIT}

DIR(cond1 → '!' cond cond2) = {'!'}

DIR(cond2 → opl cond1) = CAB'(opl) = {'||', '&&'}

DIR(cond2 → λ) = SIG(cond2) = {')', ';'}

DIR(opl → '||') = {'||'}

DIR(opl → '&&') = {'&&'}

DIR(cond → expr1 opr expr1) = CAB'(expr1) = {IDENT, '(', CONSTINT, CONSTFLOAT, CONSTLIT}

DIR(opr → '==') = {'=='}

DIR(opr → '<') = {'<'}

DIR(opr → '>') = {'>'}

DIR(opr → '>=') = {'>='}

DIR(opr → '<=') = {'<='}

DIR(restpart → '(' params ')' blq) = {'('}

DIR(params → lstparam1) = CAB'(lstparam1) = {'void', 'int', 'float'}

DIR(params → 'void') = {'void'}

DIR(lstparam1 → type IDENT lstparam2) = CAB'(type) = {'void', 'int', 'float'}

DIR(lstparam2 → ',' type IDENT lstparam2) = {','}

DIR(lstparam2 → λ) = SIG(lstparam2) = {')'}

DIR(blq → '{' sent1 sentlst '}') = {'{'}

DIR(type → 'void') = {'void'}

DIR(type → 'int') = {'int'}

DIR(type → 'float') = {'float'}

DIR(sentlst → sent1 sentlst) = CAB'(sent1) = {'void', 'int', 'float', IDENT, 'return', 'if', 'while', 'do', 'for'}

DIR(sentlst → λ) = SIG(sentlst) = {'}'}

DIR(sent1 → type lid1 ';') = CAB'(type) = {'void', 'int', 'float'}

DIR(sent1 → ident1) = CAB'(ident1) = {IDENT}

DIR(sent1 → 'return' expr1 ';') = {'return'}

DIR(sent1 → 'if' '(' cond1 ')' blq 'else' blq) = {'if'}

DIR(sent1 → 'while' '(' cond1 ')' blq) = {'while'}

DIR(sent1 → 'do' blq 'until' '(' cond1 ')') = {'do'}

DIR(sent1 → 'for' '(' IDENT '=' expr1 ';' cond1 ';' IDENT '=' expr1 ')' blq) = {'for'}

DIR(ident1 → IDENT ident2) = {IDENT}

DIR(ident2 → '=' expr1 ';') = {'='}

DIR(ident2 → '(' lexpr1 ')' ';') = {'('}

DIR(lid1 → IDENT lid2) = {IDENT}

DIR (lid2 → ',' IDENT lid2) = {',', IDENT}

DIR (lid2 → λ) = SIG (lid2) = {')'}

DIR (lexp1 → exp1 lexp2) = CAB' (exp1) = {IDENT, '(', CONSTINT, CONSTFLOAT, CONSTLIT}

DIR (lexp1 → λ) = SIG (lexp1) = {')'}

DIR (lexp2 → ',' exp1 lexp2) = {','}

DIR (lexp2 → λ) = SIG (lexp2) = {')'}

DIR (exp1 → factor1 exp2) = CAB' (factor1) = {IDENT, '(', CONSTINT, CONSTFLOAT, CONSTLIT}

DIR (exp2 → op exp1) = CAB' (op) = {'+', '-', '*', '/', '%'}

DIR (exp2 → λ) = SIG (exp2) = {'==', '<', '>', '>=', '<=', '||', '&&', ';', ')', ','}

DIR (op → '+') = {'+'}

DIR (op → '-') = {'-'}

DIR (op → '*') = {'*'}

DIR (op → '/') = {'/'}

DIR (op → '%') = {'%'}

DIR (factor1 → IDENT factor2) = {IDENT}

DIR (factor1 → '(' exp1 ')') = {'('}

DIR (factor1 → ctes) = CAB' (ctes) = {CONSTINT, CONSTFLOAT, CONSTLIT}

DIR (factor2 → '(' lexp1 ')') = {'('}

DIR (factor2 → λ) = SIG (factor2) = {'+', '-', '*', '/', '%', '==', '<', '>', '>=', '<=', '||', '&&', ';', ')', ','}

EXPLICACIÓN ERRORES CONTENIDOS EN LOS 4 CASOS DE PRUEBA

En cuanto a los casos de prueba, a continuación, explicamos los cuatro casos de prueba que tienen errores sintácticos.

1.

```
#define NUM 5
int main(void) {
    float y;
    y = 3.14;
    if (NUM > 0) {
        return x;
    else { //falta el } del if
        return y;
    }
}
```

Este caso de prueba presenta un error sintáctico en la función main, ya que el "if" no tiene llave de cierre ("}"). Por tanto, daría el siguiente error.

```
Línea 7:4 No hay alternativa viable para la entrada 'else'
    else { //falta el } del if
    ^^^^
Línea 7:4 Falta el token '['}']' y se reconoce 'else'
    else { //falta el } del if
    ^^^^
```

2.

```
int main(void) {
    int x = 5;
    float y = 3.14;
    if (x > 0) {
        return x;
    } else {
        return + y; //no reconoce el +
    }
}
// Este es un comentario de línea
/*
Esto es un comentario
de bloque.
*/
```

Este caso de prueba presenta otro error sintáctico. En este caso, el error se encuentra en la línea "int x = 5" y "float y = 3.14" ya que la gramática no permite la declaración e inicialización de variables en la misma línea. Además, hay otro error que se encuentra en el "return" del "else", ya que o sobra el "+" o falta otra variable antes del "+". Por tanto, daría el siguiente error.

```
Línea 2:10 No hay alternativa viable para la entrada '='
    int x = 5;
          ^
Línea 3:12 No hay alternativa viable para la entrada '='
    float y = 3.14;
            ^
Línea 7:15 Token no esperado '+', se esperaba [CONSTINT, IDENT, CONSTFLOAT, '(', CONSTLIT]
        return + y; //no reconoce el +
               ^
```

3.

```
#define A 2
#define 0 //falta el identificador
int main(void) {
    if (A == 0 && B == 0) {
        return 0;
    }
    else {
        return 1;
    }
}
```

Este caso de prueba presenta otro error sintáctico en el segundo “define”, ya que falta el nombre de la constante a la cual se le da el valor “0”. Por tanto, daría el siguiente error.

```
Línea 2:8 Falta el token '[IDENT]' y se reconoce '0'
#define 0 //falta el identificador
        ^
```

4.

```
int main(void) {
    int a;
    a = 2;
    return funcion_for(a);
}
void funcion_for(int) { //falta el nombre de la variable
    for (i = 0; i < num; i = i + 1) {
        num = num + 1;
    }
}
```

Este caso de prueba presenta otro error sintáctico en la segunda función, ya que falta el nombre de la variable que tiene tipo “int”. Por tanto, daría el siguiente error.

```
Línea 6:20 Falta el token '[IDENT]' y se reconoce ')'
void funcion_for(int) { //falta el nombre de la variable
                    ^
```

Cabe añadir que las capturas de pantalla anteriores de los errores se han realizado sin haber implementado aún la recuperación de errores.

Por último, mostramos los cuatro casos de prueba que son correctos.

1.

```
#define NUM 5
int main(void) {
    float y;
    y = 3.14;
    if (NUM > 0) {
        return x;
    }
    else {
        return y;
    }
}
```

2.

```
int main(void) {
    int x;
    x = 5;
    float y;
    y = 3.14;
    if (x > 0) {
        return x;
    } else {
        return y;
    }
}
// Este es un comentario de línea
/*
Esto es un comentario
de bloque.
*/
```

3.

```
#define A 2
#define B 0
int main(void) {
    if (A == 0 && B == 0) {
        return 0;
    }
    else {
        return 1;
    }
}
```

4.

```
int main(void) {
    int a;
    a = 2;
    return funcion_for(a);
}
void funcion_for(int num) {
    for (i = 0; i < num; i = i + 1) {
        num = num + 1;
    }
}
```

DESCRIPCIÓN IMPLEMENTACIÓN DE LA NOTIFICACIÓN DE ERRORES

La notificación de errores la hemos implementado basándonos en la información del libro de ANTLR (The Definitive ANTLR 4 Reference). Hemos escogido la opción llamada "TestE_Listener2", la cual subraya los errores en la salida para que al usuario le sea más sencillo localizarlos. Además, según hemos investigado, esta implementación es capaz de notificar al usuario cuando la gramática es ambigua.

Vamos a ir comentando la clase llamada "SubrayadoErrores" y detallando cuál es la función de cada clase. Esta es una subclase de "BaseErrorListener", que es una clase proporcionada por ANTLR para manejar errores durante el análisis. Se encarga de sobreescribir el método "syntaxError", el cual se llama cuando se detecta un error sintáctico durante el análisis. En este método, se imprime un mensaje de error junto con la línea y la posición del carácter donde ocurrió el error. Luego, esta misma función llama al método "subrayarError".

El método "subrayarError" se encarga de subrayar visualmente la ubicación del error en el código fuente. Obtiene el flujo de tokens ("tokens") y el código de entrada, busca la línea donde ocurrió el error y subraya la parte del código que causó el error.

Además, nos encontramos con la "ClasePrincipal", donde se encuentra el método main, que es el punto de entrada del programa. Se encarga de leer el archivo de entrada y crear un flujo de caracteres ("input") a partir de este mismo archivo, al que le pasa el analizador léxico ("GramaticaLexer"). Los tokens producidos por el analizador léxico se pasan al analizador sintáctico ("GramaticaParser"). Se establece un "SubrayadoErrores" como el listener de errores para el analizador sintáctico y un "EstrategiaError" como el manejador de errores. Finalmente, se llama al método "axioma" del analizador sintáctico para iniciar el análisis.

Por último, hemos implementado la clase "EstrategiaError", la cual extiende la clase "DefaultErrorStrategy". En esta clase sobreescribimos varios métodos:

- **reportNoViableAlternative(Parser parser, NoViableAltException e)**: Este método maneja la excepción cuando no hay una alternativa viable para la entrada. Recupera el token que causó el error y notifica los errores con un mensaje específico.
- **reportFailedPredicate(Parser parser, FailedPredicateException e)**: Este método maneja la excepción cuando una expresión no cumple con una condición de predicado definida en la gramática. Recupera la expresión del predicado que falló y notifica los errores con un mensaje específico.
- **reportInputMismatch(Parser parser, InputMismatchException e)**: Este método maneja la excepción cuando la entrada no coincide con lo que se esperaba según la gramática. Recupera el token que causó el error y los esperados, y notifica los errores con un mensaje específico.
- **reportMissingToken(Parser parser)**: Este método maneja la excepción cuando falta un token en la entrada. Recupera el token actual y los tokens esperados, y notifica los errores con un mensaje específico.
- **reportUnwantedToken(Parser parser)**: Este método maneja la excepción cuando se encuentra un token no esperado en la entrada. Recupera el token actual y los tokens esperados, y notifica los errores con un mensaje específico.

Cabe añadir que hemos implementado el método auxiliar "expectedTokens", que devuelve un conjunto de tokens esperados en el contexto actual del analizador.

En resumen, hemos implementado una notificación de errores que, si encuentra un error sintáctico en el archivo de entrada, notificará el error junto con la ubicación en el código fuente donde ocurrió el error,

subrayando visualmente la parte del código que causó el error. Además, hemos traducido los mensajes de error al español mediante la clase “EstrategiaError”.

DESCRIPCIÓN RESOLUCIÓN DE LA RECUPERACIÓN DE ERRORES

La recuperación de errores sintácticos que usa ANTLR por defecto es el sync-and-return o modo pánico. Con esta recuperación, cuando se lee un token que no coincide con el token que se debería leer en la regla, se entra en modo pánico. En modo pánico desechará los tokens hasta encontrar uno que este dentro del conjunto de resincronización (conjunto formado por la unión de los conjuntos siguientes de las reglas de la pila). Cuando se llega a este token, también desechará la pila hasta llegar a la regla de la que sale el token. De esta manera, se sincroniza y puede continuar corrigiendo la gramática. El problema del modo pánico es que puede desechar muchos tokens, por lo tanto, esta no es la primera estrategia que usa ANTLR. Las estrategias que comprueba primero son el borrado y la inserción de token.

Borrado de token. Cuando el Parser detecta un token que no es el que debería, comprueba si el token de inmediatamente después encaja en la regla. Si esto es así, borra el primer token y continua la revisión de la gramática.

Inserción de token. Si no se ha podido recuperar con el borrado de token, prueba la inserción de token. Si el token que lee es el que debería leer inmediatamente después, entonces inserta antes del token que invoca. Genera el token y no simplemente se lo salta, porque saltarse el token podría causar fallos de puntero a null en acciones de la gramática. Si no se cumple la condición anterior, entonces ahora sí pasa al modo pánico para intentar recuperarse.

Por lo tanto, la recuperación sintáctica que usa ANTLR consiste en las siguientes estrategias por orden de prioridad: borrado de token, inserción de token, modo pánico.

En cuanto a la recuperación lexica, cuando ANTLR detecte una cadena de entrada que no cumple ningún patrón de las reglas léxicas, lanzará una excepción “NoViableAltException” y continuará un lexema nuevo.