UNIVERSIDAD RAFAEL LANDÍVAR

FACULTAD DE INGENIERÍA

COMPILADORES

SECCIÓN 1 VESPERTINA

ING. MAX ALEJANDRO CERNA FLORES

TAREA 3

Julio Anthony Engels Ruiz Coto 1284719

GUATEMALA DE LA ASUNCIÓN, SEPTIEMBRE 18 DE 2024

Serie única - Gramáticas libres de contexto

Ejercicio 1:

1) Define una gramática que permita expresar condiciones con el operador ternario *condition* ? *value_if_true* : *value_if_false*

   Ejemplos de cadenas válidas:

   ```
   x ? y : z
   (var1 ? 1 : 0) ? 2 : 3
   true ? 1 : (false ? 2 : 3)
   ```

   Ejemplos de cadenas inválidas:

   `x ? y :` (falta la segunda parte del operador ternario)

   `? x : y` (falta la condición antes del ?)

   `x ? : y` (falta la primera parte de la expresión ternaria)

2) Verifica si la gramática que has propuesto es **ambigua**. ¿Cómo podrías probarlo?
3) Evalúe con una cadena propuesta por usted utilizando **Recursive Descent Algorithm,** utilice backtracking si es necesario.
4) Examina si la gramática es **recursiva por la izquierda**. Si lo es, debe proponer las modificaciones necesarias para evitar la recursividad por la izquierda.

1)
Expr → ExprTernario
ExprTernario → ExprSimple ? ExprTernario : ExprTernario | ExprSimple
ExprSimple → id | num | (Expr)

2)
Sí, la gramática es ambigua, la cadena a ? b ? c : d : e, que puede tener dos interpretaciones distintas:

(asociativa por la derecha): a ? (b ? c : d) : e

(asociativa por la izquierda): (a ? b ? c : d) : e

3)

- Tokens iniciales: (x ? y : z) ? a : b

Implementación del RDA:
Función term(TOKEN tok):

```
bool term(TOKEN tok) {
   return *next++ == tok;
}
```

Funciones para ExprSimple:

```
// ExprSimple → ID
bool ExprSimple1() {
   return term(ID);
}

// ExprSimple → NUM
bool ExprSimple2() {
   return term(NUM);
}

// ExprSimple → ( Expr )
bool ExprSimple3() {
   return term(OPEN_PAREN) && Expr() && term(CLOSE_PAREN);
}

// ExprSimple con backtracking
bool ExprSimple() {
   TOKEN *save = next;
   return (next = save, ExprSimple1()) ||
         (next = save, ExprSimple2()) ||
         (next = save, ExprSimple3());
}
```

Funciones para ExprTernario:

```
// ExprTernario → ExprSimple ? ExprTernario : ExprTernario
bool ExprTernario1() {
   return ExprSimple() &&
         term(QUESTION_MARK) &&
         ExprTernario() &&
         term(COLON) &&
         ExprTernario();
}

// ExprTernario → ExprSimple
bool ExprTernario2() {
   return ExprSimple();
}

// ExprTernario con backtracking
bool ExprTernario() {
   TOKEN *save = next;
   return (next = save, ExprTernario1()) ||
         (next = save, ExprTernario2());
}
```

Función para Expr:

```
bool Expr() {
   return ExprTernario();
}
```

Se aplicó el backtracking para decidir si después de un ExprSimple viene un ? o no.

4)

- ExprTernario → ExprTernario ? ExprTernario : ExprTernario | ExprSimple

sería recursiva por la izquierda. Para eliminar la recursividad por la izquierda, se realiza la transformación estándar:

- ExprTernario → ExprSimple ExprTernario’
- ExprTernario’ → ? ExprTernario : ExprTernario ExprTernario’ | ε

ya con esta modificación se elimina la recursividad por la izquierda y permite construir un analizador sintáctico predictivo sin backtracking.

Ejercicio 2:

1) Define una gramática que permita construir expresiones lógicas usando operadores && (AND), || (OR), y ! (NOT).

   Ejemplos de cadenas válidas:

   `a && b`

   `!a || (b && c)`

   `!(a || b) && c`

   Ejemplos de cadenas inválidas:

   `a && || b` (no se permite dos operadores consecutivos)

   `a && (b || )` (expresión incompleta dentro de los paréntesis)

   `! && a` (negación aplicada incorrectamente)

2) Verifica si la gramática que has propuesto es **ambigua**. ¿Cómo podrías probarlo?
3) Evalúe con una cadena propuesta por usted utilizando **Recursive Descent Algorithm,** utilice backtracking si es necesario.
4) Examina si la gramática es **recursiva por la izquierda**. Si lo es, debe proponer las modificaciones necesarias para evitar la recursividad por la izquierda.

1)
Expr → ExprOr
ExprOr → ExprOr || ExprAnd | ExprAnd
ExprAnd → ExprAnd && ExprNot | ExprNot
ExprNot → ! ExprNot | ExprPrimary
ExprPrimary → id | (Expr)

2)
La gramática no es ambigua.

esto pasa ya que establece claramente la precedencia y asociatividad de los operadores lógicos. Los operadores tienen las siguientes precedencias (de mayor a menor):

1. NOT (!)
2. AND (&&)
3. OR (||)

Prueba de no ambigüedad:
a || b && c.
Según la gramática, se debe interpretar como a || (b && c) debido a la mayor precedencia de && sobre ||.
Derivación única:

```
Expr
  ExprOr
          ExprOr
              ExprAnd
                   ExprNot
                        ExprPrimary (id: a)
          ||
          ExprAnd
             ExprAnd
                    ExprNot
                      ExprPrimary (id: b)
          &&
             ExprNot
                      ExprPrimary (id: c)
```

No existe otra forma válida de derivar esta expresión que contradiga la precedencia establecida, por lo que la gramática no es ambigua.

3)

- Tokens iniciales: !a && (b || !c)

Implementación del RDA:
Función term(TOKEN tok):

```
bool term(TOKEN tok) {
   return *next++ == tok;
}
```

Funciones para ExprPrimary:

```
// ExprPrimary → ID
bool ExprPrimary1() {
   return term(ID);
```

```
}

// ExprPrimary → ( Expr )
bool ExprPrimary2() {
   return term(OPEN_PAREN) && Expr() && term(CLOSE_PAREN);
}

// ExprPrimary con backtracking
bool ExprPrimary() {
   TOKEN *save = next;
   return (next = save, ExprPrimary1()) ||
         (next = save, ExprPrimary2());
}
```

Funciones para ExprNot:

```
// ExprNot → ! ExprNot
bool ExprNot1() {
   return term(NOT) && ExprNot();
}

// ExprNot → ExprPrimary
bool ExprNot2() {
   return ExprPrimary();
}

// ExprNot con backtracking
bool ExprNot() {
   TOKEN *save = next;
   return (next = save, ExprNot1()) ||
         (next = save, ExprNot2());
}
```

Funciones para ExprAnd':

```
// ExprAnd' → && ExprNot ExprAnd'
bool ExprAndPrime1() {
   return term(AND) && ExprNot() && ExprAndPrime();
}
```

```
// ExprAnd' → ε
bool ExprAndPrime2() {
   return true;
}

// ExprAnd' con backtracking
bool ExprAndPrime() {
   TOKEN *save = next;
   return (next = save, ExprAndPrime1()) ||
          (next = save, ExprAndPrime2());
}
```

Funciones para ExprAnd:

```
// ExprAnd → ExprNot ExprAnd'
bool ExprAnd() {
   return ExprNot() && ExprAndPrime();
}
```

Funciones para ExprOr':

```
// ExprOr' → || ExprAnd ExprOr'
bool ExprOrPrime1() {
   return term(OR) && ExprAnd() && ExprOrPrime();
}

// ExprOr' → ε
bool ExprOrPrime2() {
   return true;
}

// ExprOr' con backtracking
bool ExprOrPrime() {
   TOKEN *save = next;
   return (next = save, ExprOrPrime1()) ||
          (next = save, ExprOrPrime2());
}
```

Funciones para ExprOr:

```
// ExprOr → ExprAnd ExprOr'
bool ExprOr() {
   return ExprAnd() && ExprOrPrime();
}
```

Función para Expr:

```
bool Expr() {
   return ExprOr();
}
```

4)
Las producciones ExprOr y ExprAnd son recursivas por la izquierda:

- ExprOr → ExprOr || ExprAnd | ExprAnd
- ExprAnd → ExprAnd && ExprNot | ExprNot

Transformación para eliminar la recursividad por la izquierda:
Para ExprOr:

- ExprOr → ExprAnd ExprOr
- ExprOr → || ExprAnd ExprOr | ε

Para ExprAnd:

- ExprAnd → ExprNot ExprAnd
- ExprAnd → && ExprNot ExprAnd | ε

Expr → ExprOr
ExprOr → ExprAnd ExprOr'
ExprOr' → || ExprAnd ExprOr' | ε
ExprAnd → ExprNot ExprAnd'
ExprAnd' → && ExprNot ExprAnd' | ε
ExprNot → ! ExprNot | ExprPrimary
ExprPrimary → id | (Expr)

Ejercicio 3:

Define una gramática que permite construir expresiones que incluyan llamadas a funciones. Las funciones pueden tener argumentos, y las llamadas a funciones pueden estar anidadas dentro de otras expresiones.

```
func1(x, y)
func2(func1(x), y)
func3((x + y), func2(z))
```

Ejemplos de cadenas inválidas:

`func1(x, )` (falta un argumento después de la coma)

`func1(x, y` (falta el paréntesis de cierre)

`func1(x) y` (uso incorrecto fuera de la llamada a la función)

5) Verifica si la gramática que has propuesto es **ambigua**. ¿Cómo podrías probarlo?
6) Evalúe con una cadena propuesta por usted utilizando **Recursive Descent Algorithm,** utilice backtracking si es necesario.
7) Examina si la gramática es **recursiva por la izquierda**. Si lo es, debe proponer las modificaciones necesarias para evitar la recursividad por la izquierda.

1)

Expr → Expr + Term | Term
Term → FunctionCall | id | (Expr)
FunctionCall → id (ArgListOpt)
ArgListOpt → ArgList | ε
ArgList → Expr ArgListRest
ArgListRest → , ArgList | ε

2)

La gramática inicial presenta recursividad por la izquierda en la producción:

- Expr → Expr + Term | Term

Esto puede causar problemas en el análisis sintáctico recursivo descendente y potencialmente introducir ambigüedad si no se especifica la asociatividad de los operadores.

Prueba de ambigüedad:
Consideremos la expresión x + y + z.

- Interpretación 1: ((x + y) + z) (asociatividad por la izquierda)
- Interpretación 2: (x + (y + z)) (asociatividad por la derecha)

3)
Gramática sin recursividad por la izquierda:

- Expr → Term Expr'
- Expr' → + Term Expr' | ε
- Term → FunctionCall | id | (Expr)
- FunctionCall → id (ArgListOpt)
- ArgListOpt → ArgList | ε
- ArgList → Expr ArgListRest
- ArgListRest → Expr ArgListRest | ε

Cadena propuesta: func1(x + func2(y), z)

Implementación del RDA:
Función term(TOKEN tok):

```
bool term(TOKEN tok) {
   return *next++ == tok;
}
```

Funciones para Expr':

```
// Expr' → + Term Expr'
bool ExprPrime1() {
   return term(PLUS) && Term() && ExprPrime();
}

// Expr' → ε
bool ExprPrime2() {
   return true;
}

// Expr' con backtracking
bool ExprPrime() {
```

```
    TOKEN *save = next;
    return (next = save, ExprPrime1()) ||
           (next = save, ExprPrime2());
}
```

Funciones para Term:

```
// Term → FunctionCall
bool Term1() {
    return FunctionCall();
}

// Term → ID
bool Term2() {
    return term(ID);
}

// Term → ( Expr )
bool Term3() {
    return term(OPEN_PAREN) && Expr() && term(CLOSE_PAREN);
}

// Term con backtracking
bool Term() {
    TOKEN *save = next;
    return (next = save, Term1()) ||
           (next = save, Term2()) ||
           (next = save, Term3());
}
```

Funciones para Expr:

```
// Expr → Term Expr'
bool Expr() {
    return Term() && ExprPrime();
}
```

Funciones para FunctionCall:

```
bool FunctionCall() {
   return term(ID) && term(OPEN_PAREN) && ArgListOpt() && term(CLOSE_PAREN);
}
```

Funciones para ArgListOpt:

```
// ArgListOpt → ArgList
bool ArgListOpt1() {
   return ArgList();
}

// ArgListOpt → ε
bool ArgListOpt2() {
   return true;
}

// ArgListOpt con backtracking
bool ArgListOpt() {
   TOKEN *save = next;
   return (next = save, ArgListOpt1()) ||
          (next = save, ArgListOpt2());
}
```

Funciones para ArgList:

```
bool ArgList() {
   return Expr() && ArgListRest();
}
```

Funciones para ArgListRest:

```
// ArgListRest → , Expr ArgListRest
bool ArgListRest1() {
   return term(COMMA) && Expr() && ArgListRest();
}
```

```
// ArgListRest → ε
bool ArgListRest2() {
   return true;
}

// ArgListRest con backtracking
bool ArgListRest() {
   TOKEN *save = next;
   return (next = save, ArgListRest1()) ||
          (next = save, ArgListRest2());
}
```

4)

- Expr → Expr + Term | Term

Es recursiva por la izquierda. Para eliminarla, Se realiza lo siguiente:

Gramática modificada:

- Expr → Term Expr'
- Expr' → + Term Expr' | ε

Expr → Term Expr'
Expr' → + Term Expr' | ε
Term → FunctionCall | id | (Expr)
FunctionCall → id (ArgListOpt)
ArgListOpt → ArgList | ε
ArgList → Expr ArgListRest
ArgListRest → , Expr ArgListRest | ε