uc3m | Universidad Carlos III de Madrid

Procesadores del Lenguaje - 2024-2025

Grado en Ingeniería Informática

Práctica PRIMERA - Parser Descendente Recursivo

Introducción

La tarea consiste en diseñar un traductor que convierta expresiones de un lenguaje simple en notación prefija a su equivalente en notación infija.

Una expresión en notación infija como `1+2*3` se representa en notación prefija como: `(+ 1 (* 2 3))`. En el resultado de la conversión[1] los paréntesis sirven para establecer la relación de los operadores con sus respectivos operandos siguiendo la precedencia y asociatividad que definen las convenciones.

Además, los paréntesis permiten modificar el orden de evaluación predeterminado. Por ejemplo, la expresión prefija `(* (+ 1 2) 3)` se traduce a notación infija como `(1+2)*3` alterando explícitamente la precedencia estándar de las operaciones.

El traductor deberá convertir expresiones en notación prefija a su equivalente en infija:

`(+ 1 (* 2 3))` ⇒ `(1 + (2 * 3))`

`(* (+ 1 2) 3)` ⇒ `((1 + 2) * 3)`

Los paréntesis se incluyen en la salida, aunque en algunos casos no sean estrictamente necesarios, ya que garantizan que siempre se preserve el orden de evaluación previsto en la expresión.

Tareas a realizar

1. **Leed el enunciado completo.**
2. **Revisad el código proporcionado en `drLL.c`.** ¿Qué elementos léxicos proporciona `rd_lex()`?
3. **Diseñad una gramática** que represente **la sintaxis** de las expresiones aritméticas en notación prefija, siguiendo las especificaciones.
4. **Transformad la gramática** hasta que cumpla con las condiciones LL(1).
   a. **Utilizad la herramienta JFLAP** para verificar si la gramática satisface las condiciones LL(1) y para calcular los conjuntos Primero y Siguiente.
   b. **Comenzad con una gramática simplificada**, con un máximo de uno o dos operadores y un número de un solo dígito.
   c. **Expandid gradualmente la gramática**, incluyendo variables y especificaciones adicionales.
   d. **Seguid las reglas de representación**, usando letras minúsculas para terminales y mayúsculas para no terminales. **Evitad espacios en JFLAP.**
5. **Desarrollad un parser descendente recursivo**,
   a. Basado en la gramática, para reconocer expresiones en notación prefija.
   b. Incorporando semántica de traducción para generar la expresión equivalente en notación infija.
6. **Realizad pruebas exhaustivas** con un conjunto amplio de expresiones de prueba.

---

[1] La conversiones entre notaciones prefija, postfija e infija se realizan aplicando recorridos en pre-, post- e in-*order* del árbol sintáctico que representa la expresión.

Especificaciones Iniciales:

Consideraremos lo siguiente respecto al lenguaje de entrada:

- Una única[2] **`Línea de entrada`** que contendrá una **`Expresion`** terminada por **`\n`**. En la salida debe generarse la **`Expresion`** en notación infija terminada con **`\n`**
- Una **`Expresion`** puede tener:
  - una secuencia de elementos encerrada por dos paréntesis **`(Operador Parametro Parametro)`**. Para simplificar, definimos un primer elemento de tipo **`Operador`** y sólo dos elementos de tipo **`Parametro`**.
  - un elemento único de tipo **`Numero`** (o más adelante una **`Variable`**).
- Un `Operador` será alguno de entre **`+`**, **`-`**, **`*`**, **`/`**, que representan las operaciones aritméticas básicas. Lo consideraremos como un *Token.*
- Un **`Parametro`** podrá ser o bien una **`Expresion`**, o bien un `Numero`.
- Un **`Numero`** será un entero de uno o más dígitos. Lo trataremos como un *Token*.

Restricciones:

- No contemplaremos los signos unarios - y + como parte de nuestra gramática (ni a nivel sintáctico ni léxico). No podrán tratarse casos como `-1`, `-(1+2)`, o `-+2`
- El parser no deberá admitir expresiones como **`(((* 3 2)))`** o **`((* 3 2))`**. No son sintácticamente válidas.

Una vez tengáis la gramática diseñada y validada como LL(1), podéis pasar a completarla con las siguientes especificaciones:

Especificaciones Extendidas:

- **`Variables.`** Incluiremos variables de una letra, (mayúsculas o minúsculas) seguida opcionalmente de un dígito. Una **`Variable:`**
  - Estará representada por un *Token* específico.
  - Podrá intervenir en una expresión en forma de `Parametro`, al igual que un **`Numero`**. Ejemplo de traducción:
    `(+ A2 (* B 3))` ⇒ **`(A2 + (B * 3))`**
  - Podrá asignársele el valor de una **`Expresion`** mediante la expresión **`(= Variable Parametro)`**. Ejemplo de traducción:
    `(= A3 (* 2 B))` ⇒ **`(A3 = (2 * B))`**
- **Asignaciones encadenadas**: algunos lenguajes de programación permiten:
  - **`a=b=c=2+3 ;`** es válido en C.
  - **`(setq a (setq b (+ c (+ 2 3))))`** es válido en LISP, lenguaje que veremos más adelante (**`setq`** es equivalente a nuestra asignación **`=`**)

  El *parser* deberá traducir:
  `(= a1 (= b (+ c3 (+ 2 3))))` ⇒**`(a1 = (b = (c3 = (2 + 3)))`**
- **Expresiones compuestas:** Aunque pueda parecer inusual, también es posible operar en C o en LISP con las siguientes expresiones:
  - **`a = (b=2) + (c=3) ;`**
  - **`(setq a (+ (setq b 2) (setq c 3)))`**

  El *parser* deberá traducir:
  `(= a (+ (= b 2) (= c 3)))` ⇒ **`(a = ((b = 2) + (c = 3)))`**

[2] El parser se encargará de evaluar una o más expresiones gracias al bucle **`while`** en el **`main()`**.

Ejemplos de traducción:

- `A` ⇒ `A`
- `321` ⇒ `321`
- `z7` ⇒ `z7`
- `(+ A 122)` ⇒ `(A + 122)`
- `(= A (+ A 1))` ⇒ `(A = (A + 1))`
- `(+ A (* B 4))` ⇒ `(A + (B * 4))`
- `(= A (* 1231 B))` ⇒ `(A = (1231 * B))`
- `(= a (= b (+ c (+ 2 3))))` ⇒ `(a = (b = (c = (2 + 3)))`
- `(= a (+ (= b 2) (= c 3)))` ⇒ `(a = ((b = 2) + (c = 3)))`
- `(= a1 (= b (+ c3 (+ 2 3))))` ⇒ `(a1 = (b = (c3 = (2 + 3)))`

Entregas:

- **Primera Entrega:** Subid el progreso realizado el mismo día.
- **Entrega Final:** Fecha límite: **martes, 11 de marzo**.

Revisad el entregador en Aula Global para obtener indicaciones específicas.

Archivos para la Entrega Final:

1. **Archivo de código (`drLL.c`).** Incluid los datos de los autores en el encabezado:
    a. **Primera línea:** Nombres completos y número de equipo.
    b. **Segunda línea:** Correos electrónicos, separados por espacios.
2. **Memoria (`drLL.pdf`),** ~10 páginas. Debe contener (consultad la práctica guiada como referencia):
    a. ¿Qué elementos léxicos proporciona la función **`rd_lex()`**?
    b. Gramática
        i. Inicial
        ii. Final completa LL(1) que debe estar claramente identificada.
        iii. Las transformaciones aplicadas.
        iv. Distinción clara entre niveles léxico y sintáctico.
        v. Otra información relevante.
    c. Resumen del diseño del parser.
    d. Diagrama sintáctico representando la gramática diseñada.
    e. Pruebas de traducción y evaluación realizadas.
3. **Conjunto de pruebas (`drLL.txt`)** extenso y bien diseñado.

Nota Importante:

- **Dejad la entrega como borrador**, sin consolidar inmediatamente. Cualquier fallo será más fácil de corregir.
- Antes de consolidar **descargad y verificad** los archivos enviados para asegurar de que funcionan correctamente y cumplen con las instrucciones.
- **Respetad la normativa y el código ético**. Incluir el nombre de un compañero que no ha participado en la práctica será considerado fraude, lo que podría resultar en una calificación de suspenso en la evaluación continua para ambos, e incluso en otras sanciones.

Recomendaciones Generales:

- Consultad con los profesores vuestra gramática diseñada antes de proceder con el desarrollo del ***`parser`***.
- Estudiad con detenimiento el código proporcionado (**drLL.c**)
- Para limitar complicaciones podéis implementar el parser omitiendo los paréntesis en las traducciones a notación infija. Posteriormente podéis estudiar dónde incluir la impresión de los paréntesis para que la salida en infija exprese la misma operación que la entrada.
- Seguid la siguiente convención: la función **`MatchSymbol()`** asegurará que siempre esté disponible el siguiente *Token* que debe procesarse haciendo una llamada a la función **`rd_lex()`** (comprobad el código proporcionado **`drLL.c`**). Si utilizáis siempre **`MatchSymbol()`** para verificar los *Token*:
  - No será necesario incluir llamadas a la función **`rd_lex()`**
  - En algunos casos puede ser necesario preservar previamente el valor del *Token* que se verifica o acceder posteriormente a él a través de **`tokens.old_token`**
- Para evaluar las expresiones en notación prefija podéis usar el intérprete de Lisp online https://rextester.com/l/clisp:
  - Las asignaciones en Lisp se hacen cambiando **`=`** por **`setq`**
  - Para imprimir una expresión en Lisp utilizad **`(print <expresion>)`**, por ejemplo: **`(print (= a (+ 1 2)))`**

Recomendaciones de cara a la Evaluación

- La gramática debe ser lo más concisa y simple posible, manteniendo un diseño claro y bien estructurado.
- Cada función **`ParseX()`** debe:
  - llevar en comentarios las producciones que representa.
  - tener un nombre conceptualmente adecuado (igual que los No Terminales que representan).
  - coincidir con la gramática descrita en la memoria.
- El programa debe ser sencillo, bien estructurado, correctamente indentado y fácil de entender, evitando cualquier artificio o complejidad innecesaria. **Evitad**:
  - Usar variables globales adicionales a las ya proporcionadas.
  - Imprimir en cadenas de texto o archivos temporales.
  - Usar estructuras de datos adicionales como pilas.
  - El uso de bucles **`while`** o **`for`** fuera de las funciones **`main()`** o **`rd_lex()`.** Debe usarse recursividad en su lugar, siguiendo la filosofía de los *parsers* descendentes recursivos.
  - Modificar la función **`main()`**.
  - Modificar la función **`rd_lex()`**, la proporcionada es funcionalmente válida.
  - Generar ningún otro tipo de salida aparte de la expresión traducida.
- El traductor debe:
  - Generar expresiones en notación infija que sean equivalentes a la expresión en notación prefija de entrada.
  - Debe rechazar las expresiones de entrada que no sean sintácticamente válidas.

Recomendaciones para la Memoria

Debéis redactar una memoria explicando la gramática y el traductor desarrollado. Se recomienda seguir un formato similar al utilizado en las prácticas guiadas.

Especificaciones de la Memoria

- **Extensión:** Aproximadamente 10 páginas.
- **Numeración de páginas:** Comprobad que las páginas estén numeradas.
- **Identificación de los autores:** debe constar en el encabezado de la memoria.
- **Gramática:** Proporcionad una breve descripción del desarrollo de la gramática, incluyendo información relevante para el *Parser* Descendente Recursivo. Recordad que debe diferenciarse el nivel léxico del sintáctico.
- **Diseño del *Parser***: Describid de forma concisa cómo se han traducido las producciones de la gramática a código, destacando cuestiones relevantes.
- **Estilo de formato:** Seguid el formato de las prácticas guiadas, intercalando explicaciones con código y producciones, por ejemplo.
- **Diagrama Sintácticos**: se piden los diagramas que representan la gramática diseñada.
- **Pruebas:** Incluid y comentad en un anexo las pruebas diseñadas.

Recomendaciones de Presentación

- **Evitad capturas de pantalla para código y pruebas:**
  - Son difíciles de leer y pueden resultar visualmente agresivas, especialmente con fondos negros u oscuros.
  - Explorad métodos alternativos como Markdown (fácilmente generado en Visual Studio), exportación de código a RTF, o las opciones de formato integradas en Google Docs. Estas alternativas permiten hacer presentaciones más profesionales y legibles.
  - Si una captura de pantalla es necesaria (por ejemplo, para JFLAP), verificad que su contenido sea legible sin necesidad de ampliar la página.
- **Uso de colores:** es opcional, pero si se desea:
  - Algunos terminales permiten copiar y pegar código manteniendo los colores.
  - También es posible configurar esquemas de color para mejorar la legibilidad.
- **Elección de fuente:** Usad tipografías monoespaciadas como Consolas o Courier para gramáticas y código, ya que mejoran la legibilidad**.**