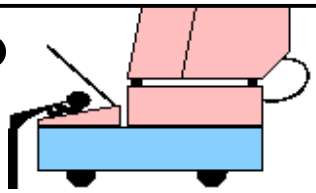
UNIVERSIDAD CARLOS III DE MADRID - DEPARTAMENTO DE INGENIERÍA TELEMÁTICA

Localización | Personal | Dc

Home / Docencia

Page Removed

Práctica 1: análisis léxico y sintáctico



Objeto de la sesión

El objetivo de esta sesión es desarrollar un analizador léxico y un analizador sintáctico de acuerdo con la especificación que se presenta más adelante. Dichos analizadores recibirán como entrada un fichero de texto que contiene el programa fuente y en el caso de que no haya errores léxicos ni sintácticos producirán como resultado un objeto Java que represente el programa fuente en forma de árbol. Si hay errores léxicos o sintácticos se levantará una excepción.

En este documento se presenta una descripción general del programa fuente que se utilizará en la práctica. Se debe notar que ciertas partes de dicha descripción no se van a utilizar en la práctica 1.

Tutoriales

Antes de empezar a realizar la práctica se recomienda consultar los manuales de CUP y JLex disponibles a través de Aula Global o bien estos pequeños resúmenes de CUP y JLex

Definición del lenguaje de programación a traducir

Descripción general del lenguaje de programación fuente PF2016

Empecemos con un pequeño programa de ejemplo:

```
PF2016 Ejemplo(argumento)

vars

int x, y;

{
x:=Str2Int(argumento);
y:=0;

while(x>0) {
 y:=y+x;
 x:=x-1;
}

print("Resultado= " + Int2Str(y));

}
```

Un programa en el lenguaje fuente consta de las siguientes partes:

1. Cabecera del programa. En el ejemplo anterior la cabecera es la línea:

   ```
   PF2016 Ejemplo(argumento)
   ```

   Consta de: palabra clave `PF2016`, seguida del nombre del programa (en el ejemplo `Ejemplo`), seguida opcionalmente de uno o varios identificadores separados por comas entre paréntesis. Estos identificadores son variables de tipo `string` que se utilizan para que el programa pueda recibir argumentos al ser ejecutado. Cuando se invoca el programa, lo que se escriba a continuación del

nombre del programa se le pasará al programa en estas variables.

2. Opcionalmente, pueden declararse variables que serán utilizadas dentro del programa. Las declaraciones de variables están formadas por un tipo de datos seguido de una lista de identificadores (nombres de variables). Los tipos de datos existentes en este lenguaje de programación son `int`, `bool` y `string`.

   Cuando se declara una variable se supondrá que inicialmente tomará el valor `false` si la variable es de tipo `bool`, 0 si la variable es de tipo `int` y `""` si la variable es de tipo `string`.

3. Cuerpo del programa, formado por una secuencia de sentencias que describen las operaciones que realiza el programa.

Las sentencias de un programa pueden ser de los siguientes tipos:

- Sentencia de asignación: almacena en la variable indicada a la izquierda del `:=` el valor que se obtiene como resultado de evaluar la expresión a la derecha del `:=`.
- Sentencia condicional `if <Exp> then <Sent> [[thenx <Sent>] else <Sent>] endif`. Los corchetes indican partes opcionales. La expresión de la parte `if` de la sentencia es de tipo `bool`. En este lenguaje los datos de tipo `bool` pueden tomar 3 valores: `true`, `false` o `unk`. Cuando el valor de la expresión sea `true` se ejecuta la parte `then` de la sentencia. Si la sentencia tiene parte `thenx` y el valor de la expresión es `unk` se ejecuta dicha parte. Finalmente, la parte `else` de la sentencia, si existe, se ejecuta si la expresión se evalúa a `false` o si la expresión se evalúa a `unk` y la sentencia no tiene parte `thenx`.
- Bucle `while <Exp> <Sent>`. Se ejecuta el bucle mientras la expresión se evalúe a `true`.
- Sentencia `print ( <Exp> )`. Esta sentencia recibe como argumento un dato de tipo `string` que imprime en pantalla, seguido de un salto de línea.

Especificación del analizador léxico

El analizador léxico JLex a desarrollar debe reconocer los siguientes tokens:

- Palabras clave: `and` (token `AND`), `or` (token `OR`), `not` (token `NOT`), `PF2016` (token `PROG`), `vars` (token `VARS`), `while` (token `WHILE`), `print` (token `PRINT`), `thenx` (token `THENX`), `Int2Str` (token `INT2STR`),`Str2Int` (token `STR2INT`), `if` (token `IF`), `then` (token `THEN`), `else` (token `ELSE`), `endif` (token `ENDIF`), `int` (token `TIPO`), `bool` (token `TIPO`), `string` (token `TIPO`)
- Signos de puntuación y operadores: ";" (token `PC`), ":=" (token `ASOP`), "+" (token `MAS`), "-" (token `MENOS`), "*" (token `POR`), "/" (token `DIV`), ">" (token `MAYORQUE`), "<" (token `MENORQUE`), "=" (token `IGUALQUE`), "{" (token `ABRELLAVE`), "}" (token `CIERRALLAVE`), "," (token `COMA`), "(" (token `PAREN`), ")" (token `TESIS`).
- Identificadores (token `IDENT`): comienzan por una letra (mayúscula o minúscula), seguida opcionalmente de cualquier cadena de letras y dígitos
- Constantes, que pueden ser de tres tipos:
  - Números enteros no negativos (token `CENT`), es decir, cualquier cadena de dígitos. Ejemplos: `1`, `0`, `2010`
  - Las constantes lógicas (token `CLOG`) `true`, `false` y `unk`
  - Constantes de tipo string (cadenas de caracteres) (token `CST`). Necesariamente tienen que comenzar y terminar con el carácter comilla (`"`) y entre las dos comillas puede haber cualquier cadena de caracteres que no contenga el símbolo comilla.

El lenguaje distingue mayúsculas y minúsculas; por lo tanto mientras `print` es una palabra clave, `Print` es un identificador y `PRINT` es otro identificador distinto del anterior.

Recuerde que también tiene que especificar en el fichero JLex que se deben reconocer y consumir (sin generar token) los espacios en blanco, tabuladores, saltos de línea y retornos de carro.

Para desarrollar el analizador léxico pedido lo único que hay que hacer es completar este esqueleto de fichero JLex en el que lo único que falta es definir para cada token a reconocer, la expresión regular asociada

y la acción asociada. A modo de ejemplo, se proporciona ya definido el token para la palabra clave `and`.

Como puede observarse en la definición de la regla para la palabra clave `and`, lo que hay que hacer es:

1. Para cada token definir la expresión regular asociada, según el formato requerido por JLex (véase la documentación que se proporciona).
2. A continuación de la expresión regular definir la acción que se toma cuando se reconoce dicho token. Para los tokens que no tienen ningún dato asociado, escribiremos `{return tok(sym.X, null); }`, donde `X` es el nombre que se le ha dado al token en cuestión en el fichero CUP.

   Para los tokens que tienen un dato asociado, escribiremos `{return tok(sym.X, obj); }`, donde `X` es el nombre que se le ha dado al token en cuestión en el fichero CUP y `obj` es el objeto Java asociado al token. Para construir el objeto Java asociado a un token puede utilizar el método `yytext()`, tal y como se explica en la documentación de JLex.

   Como se explica en el manual de CUP y se ve en clase, un fichero CUP declara una lista de tokens, donde cada token es representado por un idendificador (por ejemplo, `AND`). Cuando se genera el analizador sintáctico a partir del fichero CUP, se producen como resultado 2 ficheros Java. Uno de ellos, que lleva por nombre `sym.java` contiene una clase llamada `sym` en la que simplemente se asocia un número entero a cada uno de los tokens que se han declarado en el fichero CUP. Por lo tanto, `sym.X` es el entero que utiliza la clase `sym` para identificar al token `X`.

3. En el caso de los espacios en blanco, saltos de línea, etc., escribiremos como acción `{ }`, cuyo efecto es no devolver nada y seguir consumiendo caracteres de entrada para producir el siguiente token.

El analizador léxico deberá lanzar una excepción de tipo `LexerException`, que se proporciona, cuando se detecte un error léxico.

Definición de la sintaxis del lenguaje de programación fuente

A continuación vamos a dar la definición de la sintaxis del lenguaje fuente por medio de una gramática independiente del contexto (excepto para las expresiones, que se van a definir en lenguaje natural).

La gramática que define la sintaxis del lenguaje de programación fuente es la siguiente:

```
G=<ET, EN, S, P>

ET= {PC, ASOP, IF, THEN, ELSE, ENDIF, WHILE, PRINT, INT2STR, STR2INT,
THENX, NOT, MENORQUE, MAS, MENOS, UMENOS, POR, DIV, AND, OR, IGUALQUE,
MAYORQUE, PAREN, TESIS, PROG, ABRELLAVE, CIERRALLAVE, COMA, VARS, TIPO,
CENT, CLOG, CST, IDENT}

EN= {S, <LVar>, <Sent>, <Body>, <SentSimp>, <Asign>, <Cond>,
    <Repet>, <Print>, <Exp>, <VDef>, <Decl>}

S= S

P= {
S -> PROG IDENT PAREN <LVar> TESIS <Body>
       | PROG IDENT <Body>
       | PROG IDENT VARS <VDef> <Body>
       | PROG IDENT PAREN <LVar> TESIS VARS <VDef> <Body>

<VDef> -> <Decl> PC
       | <Decl> PC <VDef>

<Decl> -> TIPO <LVar>

<LVar> -> IDENT
     |   IDENT COMA <LVar>

<Body> -> <Sent>

<SentSimp> -> <Asign> PC
        |   <Cond>
```

```
        |   <Repet>
        |   <Print> PC

<Print>-> PRINT PAREN <Exp> TESIS

<Asign>-> IDENT ASOP <Exp>

<Cond>-> IF <Exp> THEN <Sent> ENDIF
    |   IF <Exp> THEN <Sent> ELSE <Sent> ENDIF
    |   IF <Exp> THEN <Sent> THENX <Sent> ELSE <Sent> ENDIF

<Repet>-> WHILE <Exp> <Sent>
}
```

NOTA: no se proporcionan, intencionadamente, reglas de producción que tengan a `<Sent>` como antecedente. Deberá definirlas usted (puede ser necesario añadir algún símbolo no terminal auxiliar). `<Sent>` representa una sentencia o bien una lista de una o más sentencias entre llaves.

Expresiones en el lenguaje de programación fuente

Las expresiones del lenguaje de programación fuente (identificadas en la definición BNF con la variable de lenguaje `<Exp>`) deben ajustarse a lo siguiente:

Expresiones sin tipo definido:

- Un identificador que sea el nombre de una variable. Será del tipo de la variable.
- Una expresión entre paréntesis. Será del tipo de la expresión contenida dentro de los paréntesis.

Expresiones de tipo `int`:

- Una constante expresada en decimal.
- Una suma, resta, multiplicación o división de expresiones de tipo `int` utilizando los operadores habituales "+", "-", "*", "/".
- El opuesto de una expresión de tipo `int`, utilizando el símbolo habitual "-" antes de la expresión.
- Una expresión que pertenezca al lenguaje definido por la expresión lingüística `Str2Int "(" <Exp> ")"`, donde la subexpresión entre paréntesis ha de ser de tipo `string`. El resultado de evaluar esta expresión será el número entero que se obtendría en Java al invocar el método `Integer.parseInt` sobre el dato que se obtendría al evaluar la subexpresión de tipo `string`.

Expresiones de tipo `string`:

- Una constante que se representa como cualquier cadena de símbolos entre comillas ("), donde la cadena de símbolos no debe de contener la comilla.
- Concatenaciones de expresiones de tipo `string` utilizando el operador "+".
- Una expresión que pertenezca al lenguaje definido por la expresión lingüística `Int2Str "(" <Exp> ")"` , donde la subexpresión entre paréntesis ha de ser de tipo `int`. El resultado de evaluar esta expresión será una cadena de caracteres que represente el número entero que se obtendría al evaluar la subexpresión de tipo `int` en base decimal.

Expresiones de tipo `bool`:

- Las constantes `true`, `unk` y `false`.
- La conjunción o disyunción de expresiones lógicas, utilizando los operadores `and` y `or` respectivamente.
- Una expresión lógica negada, utilizando el operador `not` antes de la expresión lógica.

- La igualdad entre 2 expresiones utilizando el operador "=". Las dos expresiones comparadas deberán de ser del mismo tipo.
- La comparación entre 2 expresiones de tipo `int` utilizando los operadores "<" (menor que) o ">" (mayor que).

Reglas de precedencia

La precedencia de los operadores es, de menor a mayor:

1. `or`
2. `and`
3. `not`
4. =
5. < y >
6. + y -
7. * y /
8. - (opuesto)

Todos los operadores asocian por la `derecha`.

Requisitos que deben cumplir los ficheros JLex y CUP

Para la prueba del analizador léxico y el analizador sintáctico, se utilizara la clase Main que se proporciona. La ejecución de dicha clase se realizará de acuerdo con lo siguiente:

```
java Main <nombre_fichero>
```

Donde `<nombre_fichero>` es el nombre del fichero (con el path si es necesario) del programa fuente a analizar.

El programa deberá de levantar una excepción que no deberá de ser capturada en el caso de que el programa fuente tenga algún error de sintaxis. Los mensajes emitidos por excepciones producidas por errores en la fase de análisis sintáctico deberán indicar una línea del programa orientativa del lugar en el que se ha producido el error.

Asímismo, el programa deberá de levantar una excepción de tipo `LexerException`, que no deberá de ser capturada en el caso de que el programa fuente tenga algún error léxico.

Obligatoriamente las clases generadas por CUP deberán pertenecer a un paquete llamado Parser y la clase generada por JLex deberá pertenecer a un paquete llamado Lexer.

Orden de compilación

Antes de entregar la práctica deberá cerciorarse de que su práctica puede compilarse correctamente con `javac` siguiendo el siguiente orden:

1. Clases del paquete `Errors`.
2. Clases del paquete `AST` que usted debe desarrollar.
3. Clases `parser` y `sym` generadas por CUP.
4. Clase `Yylex` generada por JLex.
5. Clase `Main` que se proporciona.

Aquellas prácticas que no se puedan compilar en este orden serán calificadas con 0.

Ayudas y sugerencias

Se proporciona un juego de tests con el que probar la práctica. De entre ellos solamente deberían de producir un error de sintaxis los tests en carpetas cuyo nombre comienza por `ErrSint` y error léxico los tests en

carpetas cuyo nombre comienza por `ErrLex`. Puede ocurrir que alguno de los ejemplos `ErrLex` de error de sintaxis (y no léxico).

Se advierte que se realizarán tests adicionales a los proporcionados a las prácticas recibidas, por lo que se recomienda a los alumnos que planifiquen tests complementarios a su práctica.

Ficheros a entregar

Se deberán entregar exclusivamente los siguientes ficheros:

- fichero `Yylex` en formato JLex para el analizador léxico
- fichero `parser` en formato CUP para el analizador sintáctico
- fichero `AST.zip`, que contiene exclusivamente un directorio de nombre `AST` que a su vez contiene exclusivamente los ficheros `.java` que usted ha desarrollado para modelar árboles de sintaxis abstracta.

Localización | Personal | Docencia | Investigación | Novedades | Intranet
inicio | mapa del web | contacta

Last Revision: 02/24/2016 20:10:51