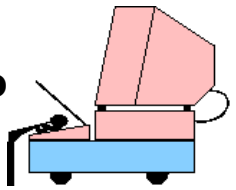
Práctica 1: análisis léxico y sintáctico



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 PF2013

Empecemos con un pequeño programa de ejemplo:

```
state_machine

states s1, s2, s3;

final states s4;

inputs a, b, c;

outputs sal;

starts with s1;

sal:='0';

begin

transition(s1,a,s2);

transition(s2,c,s1);

transition(s2,b,s3);

transition(s3, c, s4);

behaviour
sal:='1';

transition(s3, a, s2);

transition(s3, b, s1);

end
```

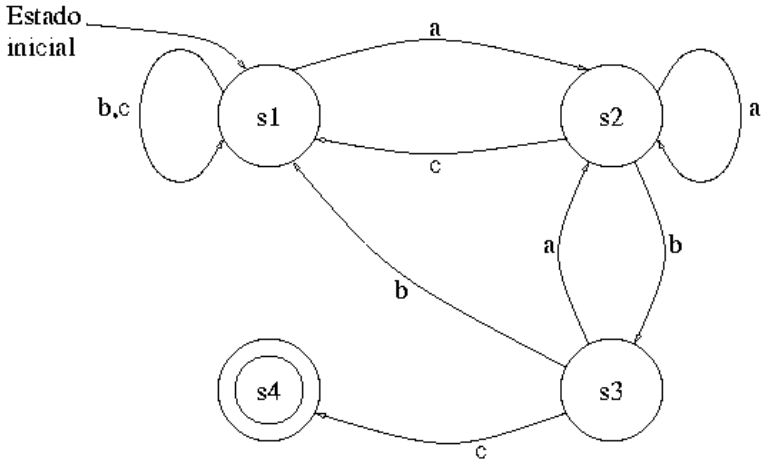
Un programa en el lenguaje fuente consta de las siguientes partes:

1. Palabra clave `state_machine` que indica el inicio del programa.
2. Declaración de estados de la máquina de estados, donde cada estado es un identificador. Comienza con la palabra clave `states`.
3. Opcionalmente, declaración de estados finales que serán distintos de los estados declarados en el apartado anterior. Comienza con las palabras clave `final states`.
4. Declaración de eventos de entrada a la máquina de estado, donde cada evento es un identificador. Comienza con la palabra clave `inputs`.
5. Opcionalmente, declaración de variables locales. No será necesario declarar tipo, porque todas ellas serán de tipo `boolean`. Comienza con la palabra clave `local`.
6. Declaración de variables de salida. No será necesario declarar tipo, porque todas ellas serán de tipo `boolean`. Comienza con la palabra clave `outputs`.
7. Declaración del estado inicial de la máquina de estados. Comienza con las palabras clave `starts with`.
8. Inicialización de las variables locales y de salida.
9. Definición de las transiciones de la máquina de estados. Comienza con la palabra clave `begin` y termina con la palabra clave `end`. Cada transición

contine el estado inicial, el evento que dispara la transición y el estado final. Opcionalmente, una transición puede contener:

- Una condición que se debe cumplir para que se dispare la transición. La condición es una expresión booleana que se debe evaluar a `'1'` para que la transición se dispare.
- Una lista de sentencias (precedida de la palabra clave `behaviour`). Las sentencias pueden ser de los siguientes tipos: asignación de expresiones a variables locales o de salida, sentencia condicional `if` y sentencia `stop` (termina la ejecución).

La siguiente figura representa gráficamente la máquina de estados definida por el programa anterior. Debe tenerse en cuenta que, para aquellas transiciones que no se han definido se supone que por defecto se permanece en el mismo estado.



Especificación del analizador léxico

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

- Palabras clave: `begin` (token `BEGIN`), `end` (token `END`), `and` (token `AND`), `or` (token `OR`), `not` (token `NOT`), `state_machine` (token `STATE_MACHINE`), `states` (token `STATES`), `final` (token `FINAL`), `inputs` (token `INPUTS`), `local` (token `LOCAL`), `outputs` (token `OUTPUTS`), `stop` (token `STOP`), `condition` (token `CONDITION`), `transition` (token `TRANSITION`), `behaviour` (token `BEHAVIOUR`), `starts` (token `START`), `with` (token `WITH`), `if` (token `IF`), `then` (token `THEN`) y `endif` (token `ENDIF`).
- Signos de puntuación y operadores: ";" (token `PC`), ":=" (token `ASIG`), "(" (token `PAREN`), ")" (token `TESIS`) y "," (token `COMA`).
- Identificadores (token `IDENT`): comienzan por una letra (mayúscula o minúscula), seguida opcionalmente de cualquier cadena formada por letras, dígitos.y el carácter "_".
- Las constantes lógicas '1' y '0' (token `CLOG`)

El lenguaje distingue mayúsculas y minúsculas; por lo tanto mientras `begin` es una palabra clave, `Begin` es un identificador y `BEGIN` 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= {STATE_MACHINE, COMA, PC, PAREN, TESIS, BEGIN, END, ASIG, STATES,
```

```
    FINAL, INPUTS, LOCAL, OUTPUTS, TRANSITION, BEHAVIOUR, AND, OR, NOT,
    START, WITH, CONDITION, STOP, IF, THEN, ENDIF, IDENT, CLOG}

EN= {<S>, <StateDeclList>, <FinalStateDeclList>, <IdentList>, <InputEventDecl>, <LocalVarDecl>,
    <OutputEventDecl>, <Transitions>, <TransitionList>, <TransBehaviour>, <Statement>, <Transition>, <InitialState>,
    <Event>, <FinalState>, <ExpLog>, <Condition>, <Initialization>, <SimpleStatementList>}

S= <S>

P= {
<S> ->  STATE_MACHINE <StateDeclList> <FinalStateDeclList> <InputEventDecl> <LocalVarDecl> <OutputEventDecl> <Initialization> <Transitions>
   |  STATE_MACHINE <StateDeclList> <FinalStateDeclList> <InputEventDecl> <OutputEventDecl> <Initialization> <Transitions>
   |   STATE_MACHINE <StateDeclList> <InputEventDecl> <LocalVarDecl> <OutputEventDecl> <Initialization> <Transitions>
   |  STATE_MACHINE <StateDeclList> <InputEventDecl> <OutputEventDecl> <Initialization> <Transitions>

<StateDeclList> -> STATES <IdentList> PC

<FinalStateDeclList> -> FINAL STATES <IdentList> PC

<IdentList> -> IDENT
   | IDENT COMA <IdentList>

<InputEventDecl> -> INPUTS <IdentList> PC

<LocalVarDecl> -> LOCAL <IdentList> PC

<OutputEventDecl> -> OUTPUTS <IdentList> PC

<Transitions> -> BEGIN <TransitionList> END

<TransitionList> -> <Transition>
   | <Transition> <TransitionList>

<Transition> -> TRANSITION PAREN <InitialState> COMA <Event> COMA <FinalState> TESIS PC
   | TRANSITION PAREN <InitialState> COMA <Event> COMA <FinalState> TESIS PC <TransBehaviour>
   | TRANSITION PAREN <InitialState> COMA <Event> COMA <FinalState> TESIS PC <Condition>
   | TRANSITION PAREN <InitialState> COMA <Event> COMA <FinalState> TESIS PC <Condition> <TransBehaviour>

<Condition> -> CONDITION PAREN <ExpLog> TESIS PC

<InitialState> -> IDENT

<Event> -> IDENT

<FinalState> -> IDENT

<TransBehaviour> -> BEHAVIOUR <Statement>

<Statement> -> <Statement> <Statement>
   | STOP PC
   | IDENT ASIG <ExpLog> PC
   | IF <ExpLog> THEN <Statement> ENDIF PC

<Initialization>  -> START WITH IDENT PC <SimpleStatementList>

<SimpleStatementList> -> <SimpleStatementList> IDENT ASIG CLOG PC
   | IDENT ASIG CLOG PC
}
```

Expresiones en el lenguaje de programación fuente

Las expresiones lógicas del lenguaje de programación fuente (identificadas en la gramática por el símbolo no terminal `<ExpLog>`) deben de ajustarse a lo siguiente:

- Las constantes lógicas pertenecen al lenguaje `<ExpLog>`.
- Los identificadores pertenecen al lenguaje `<ExpLog>`.
- Una expresión lógica entre paréntesis pertenece al lenguaje `<ExpLog>`.
- La conjunción o disyunción de expresiones lógicas, utilizando los operadores `and` y `or` respectivamente pertenecen al lenguaje `<ExpLog>`.
- Una expresión lógica negada, utilizando el operador `not` antes de la expresión lógica pertenece al lenguaje `<ExpLog>`.

Reglas de precedencia

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

1. `or`
2. `and`
3. `not`

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.

Last Revision: 01/23/2015 20:08:29