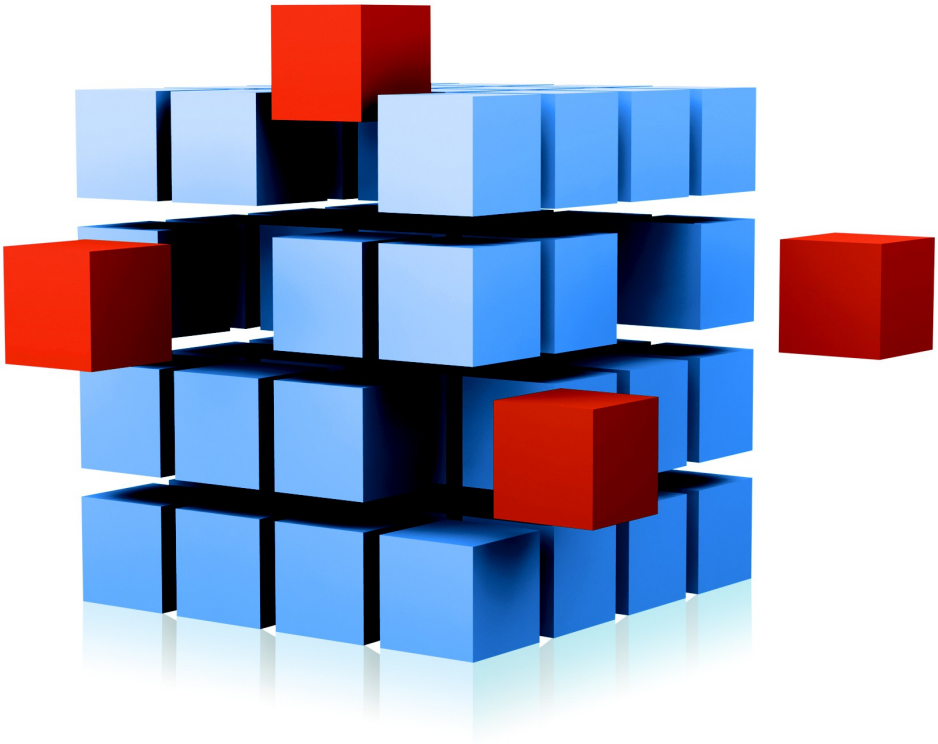


MiniJava Compiler

Compilador completo (Manual de Usuario) – 2014

Integrantes:

Brenda Soledad Dilschneider y Cuenca Francisco.

BRENDA S. DILSCHNEIDER

Cuenca Francisco

Índice.

BRENDA S. DILSCHNEIDER

Cuenca Francisco

Introducción.

En el informe actual se presentará el Manual de Usuario correspondiente al Compilador de MiniJava realizado sobre las pautas presentadas por la cátedra de *Compiladores e Intérpretes 2013*.

En el mismo se mostrará toda la información de interés para el programador que utilice el compilador junto al intérprete CEeIVM (provisto por la cátedra). Se incluirá una especificación del lenguaje aceptado por el compilador. Finalizando con una guía sobre cómo ejecutar el compilador y el intérprete CeIVM y decisiones de diseño tomadas para la implementación del mismo.

BRENDA S. DILSCHNEIDER

Cuenca Francisco

Especificación del lenguaje.

Para dar el lenguaje sobre sobre el que se desarrolló el compilador, primero se mostrará la EBNF sobre la que el mismo se basó explicando aquellas producciones que sean dignas de requerir dicha explicación.

<Inicial> → <Clase>$^{+}$ Puede declararse más de una clase, pero como mínimo una.

<Clase> → **class** identificador <Herencia>$^{?}$ { <Miembro>* }
La declaración de una clase puede tener o no herencia (pudiendo heredar solo de una clase). Y puede tener un cuerpo, definido como Miembro, o el mismo puede no estar, teniendo en este caso una clase vacia.

Ejemplo con Herencia:
class A extends B { }

Ejemplo sin Herencia:
class A {}

<Herencia> → **extends** identificador

<Miembro> → <Atributo> | <Ctor> | <Metodo>
Un Miembro puede ser la declaración de un atributo, de un constructor o de un método.

<Atributo> → **varinst** <Tipo> <ListaDecVars> **;**
Un atributo es declarado comenzando con la palabra reservada "*varinst*", seguida del tipo, una lista de variables y finaliza con el carácter ";".

Ejemplo con lista de variables de número mayor a uno:
varinst int a,b,c ;

<Metodo> → <ModMetodo> <TipoMetodo> identificador <ArgsFormales> <Bloque>
La declaración de un método consta de un modificador seguido del tipo del método, su identificador, los argumentos formales, es decir, aquellos correspondientes a la declaración, y el cuerpo del mismo, denominado Bloque.

Ejemplo:

```
static int invertir(int numero, int digitos)
    {
    varlocal int cuenta, aux;
            cuenta = 0;
            for(aux = 0; aux < digitos; aux = aux + 1) {
                    cuenta = cuenta * 10 + numero % 10;
                    numero = numero / 10;
            }

            return cuenta;                  }
```

<Ctor> → identificador <ArgsFormales> <Bloque>
Un constructor se declara de la misma manera que un método, sólo que en este caso, no se cuenta con un modificador ni un tipo. En un constructor, no pueden haber sentencias return, como si hay en el ejemplo anterior.

<ArgsFormales> → **(**<ListaArgsFormales>$^{?}$**)**
Los argumentos formales de un método o un constructor son declarados como una lista de argumentos formales, la cual puede, o no, estar presente. La misma es encerrada entre los caracteres "(" y ")".

Ejemplo sin lista de argumentos formales:

```
static int invertir(){
    return 0;
}
```

En el ejemplo para la declaración de método, se encuentra la otra declaración posible para los argumentos formales.

<ListaArgsFormales> → <ArgFormal>
<ListaArgsFormales> → <ArgFormal> , <ListaArgsFormales>
Una lista de argumentos formales está constituida por muchos argumentos formales separados por el carácter ",", o sólo por un argumento formal.

<ArgFormal> → <Tipo> identificador
Un argumento formal se declara con su tipo seguido de su identificador asociado.

Ejemplo:
int a

<ModMetodo> → **static** | **dynamic**
El modificador de un método está constituido o bien por la palabra reservada "static" o bien por la palabra "dynamic", no puede ser otra palabra.

<TipoMetodo> → <TipoPrimitivo> | **void**
El tipo de un método puede ser void o de tipo "Tipo" (definido más adelante).

<Tipo> →<TipoPrimitivo> | identificador

Un tipo puede ser un Tipo Primitivo o un identificador correspondiente al nombre de una clase, es decir, un Tipo Clase.

<TipoPrimitivo> → **boolean** | **char** | **int** | **String**
Un tipo primitivo está dado por las palabras reservadas “boolean”, “char”, “int” o “String”, cualquier otra palabra no será considerada un tipo primitivo.

<ListaDecVars> → identificador | identificador **,** <ListaDecVars>
Una declaración de una lista de variables está dada por un solo identificador, o por muchos identificadores separados por el carácter “,”.<Bloque> → { <Sentencia> *}
Un bloque está dado por cero o más sentencias encerradas entre los caracteres “{” al principio y “}” al final.

Un ejemplo de bloque con mas de una sentencia se muestra en el ejemplo dado en la declaración de Método.

<Sentencia> → **;**
<Sentencia> → <Asignacion>
<Sentencia> → <SentenciaSimple> **;**
<Sentencia> → **if (**<Expresion>**)** <Sentencia>
<Sentencia> → **if (**<Expresion>**)** <Sentencia> **else** <Sentencia>
<Sentencia> → **while (**<Expresion>**)** <Sentencia>
<Sentencia> → **for (**<Asignacion> **;** <Expresion> **;** <Asignacion>**)** <Sentencia>
<Sentencia> → <Bloque>

<Sentencia> → **return** <Expresion> **;** | **return ;**

La declaración de Sentencias está dada por sólo un carácter “;”, una asignación, una sentencia simple finalizada en “;”, los dos tipos de sentencias if posibles, es decir, con y sin el “else”, la sentencia “while”, la sentencia “for” o la sentencia “return” siguiendo la especificación recién mostrada.

Ejemplos de sentencias:

```
return;
```

```
if(a == 2)
    b = true;
    else b = false;
```

```
a = 1;
while(a < 10){
    b = b * 10 + a;
    a = a +1;
}
```

```
for (i = 0; i<10; i=i+1){
    b = b * 10 + i;
}
```

<Asignacion> → <LadoIzquierdo> = <Expresion>
La declaración de una asignación está dada por un <LadoIzquierdo> (definido a continuación) seguido del carácter "=", y este, a su vez es seguido de una expresión.

Ejemplo:
a=2;

<LadoIzquierdo> → **identificador** <idEncadenado>*

<idEncadenado> → . **identificador**

Un lado izquierdo corresponde a un encadenado de identificadores, osea una secuencia de ids separadas por un punto

Ejemplo:

a.b.c

<SentenciaSimple> → **(**<Expresion>**)**
Una Sentencia Simple esta dada por una Expresión encerrada entre los caracteres "(" y ")".

Ejemplo:
(2*3 + 4)

<Expresion> → <Expresion> <OperadorBinario> <Expresion>
<Expresion> → <OperadorUnario> <Expresion>
<Expresion> → <Operando>

La declaración de una expresión está dada por una expresión seguida de un operador binario el cual es a su vez seguido de un operador binario, o por un operador unario seguido de una expresión o por un Operando -+6/(definido mas adelante).

Ejemplos:
3*3
!(2 == 2)
new Clase()

<OperadorBinario> → || | && | == | != | < | > | <= | >= | + | - | * | / | %
Un operador binario consta de alguno de los caracteres definidos aquí arriba.

<OperadorUnario> → ! | + | -
Un operador unario consta de alguno de los caracteres definidos aquí arriba.

<Operando> → **this**
<Operando> → <Literal>

<Operando> → <Primario>

Un operando puede ser, un objeto actual (this), un literal o un primario (definidos a continuación)

<Primario> → **(** <Expresion> **)** <LlamadaoIdEncadenado>*
<Primario> → identificador <LlamadaoIdEncadenado>*
<Primario> → **new** identificador <ArgsActuales> <LlamadaoIdEncadenado>*
<Primario> → identificador <ArgsActuales> <LlamadaoIdEncadenado>*
Un primario está dado por una expresión encerrada entre los caracteres "(" y ")" seguida de cero o más llamadas o identificadores encadenados, o por un identificador seguido de identificadores encadenados , o por la palabra reservada "new" seguida de un identificador, argumentos actuales y cero o más llamadas o identificadores encadenados , o, por último, por un identificador seguido de argumentos actuales y estos a su vez, seguidos por cero o más llamadas o identificadores encadenados.
La definición de llamada será dada al concluir con los ejemplos de Primario.

Ejemplos:
(this).metodo()
a.metodo()
new Clase(true,2)
metodo(true,2).m2().a

<LlamadaoIdEncadenado> → **.**identificador <ArgsActuales>

<LlamadaoIdEncadenado> → **.**identificador

Una llamada o identificador encadenado puede ser, un identificador directo o bien un identificador seguirdo de argumentos (método).

Ejemplo:

.a

.a(arg1,arg2)

<Literal> → **null** | **true** | **false** | **intLiteral** | **charLiteral** | **stringLiteral**
Un literal se declara como las palabras reservadas “null”, “true” o “false” o como un Literal de tipo entero, char o String.

<ArgsActuales> → **(** <ListaExps>? **)**
La declaración de los Argumentos Actuales esta dada por cero o una Lista de Expresiones encerrada entre los caracteres “(” y “)”.

Ejemplos:
(arg1,arg2)

<ListaExps> → <Expresion> | <Expresion> **,** <ListaExps>
Una lista de expresiones esta dada por solo una expresión o por muchas expresiones separadas por el carácter “,”.

Desiciones de diseño.

A continuación se describe las decisiones de diseño tomadas.

- Se optó por modificar la sentencia “For” de la siguiente manera:
  Original:

  <Sentencia> → for (<Asignacion> ; <Expresion> ; <Expresion>) <Sentencia>

  Modificada:

  <Sentencia> → for (<Asignacion> ; <Expresion> ; <Asignacion>) <Sentencia>

De esta forma se le da mayor expresividad al lenguaje ya que el programador tiene la posibilidad de controlar cómo evoluciona la variable de control.

- Se admite que mas de una clase defina un método “main”, ejecutándose el primero encontrado. Sin embargo, se arrojará por pantalla una advertencia informando que se ha declarado mas de un método “main” y qué clases lo hicieron.

Los comentarios siguen la forma presentada en la descripción de la sintaxis de MiniJava, es decir, se siguió el siguiente enunciado:

"/* comentario */ Comentarios multi-línea: Todos los caracteres desde /* hasta */ son ignorados."

Se decidió el uso de la estructura ArrayList para almacenar las palabras reservadas debido a las facilidades que provee Java para el manejo de la misma.

Otra decisión de diseño que se tomó es utilizar un switch para el modelado del Autómata debido a que las búsquedas en esta estructura es muy eficiente al ser la misma asociativa.

Se tomó la decisión de aceptar la cadena vacía como un literal String, por ejemplo: “”

La gramática obtenida como resultado de las transformaciones de factorización y eliminación de recursividad a izquierda, no es de tipo LL(1) ya que es ambigua. La ambigüedad se da en las siguientes reglas de producción:

<Se

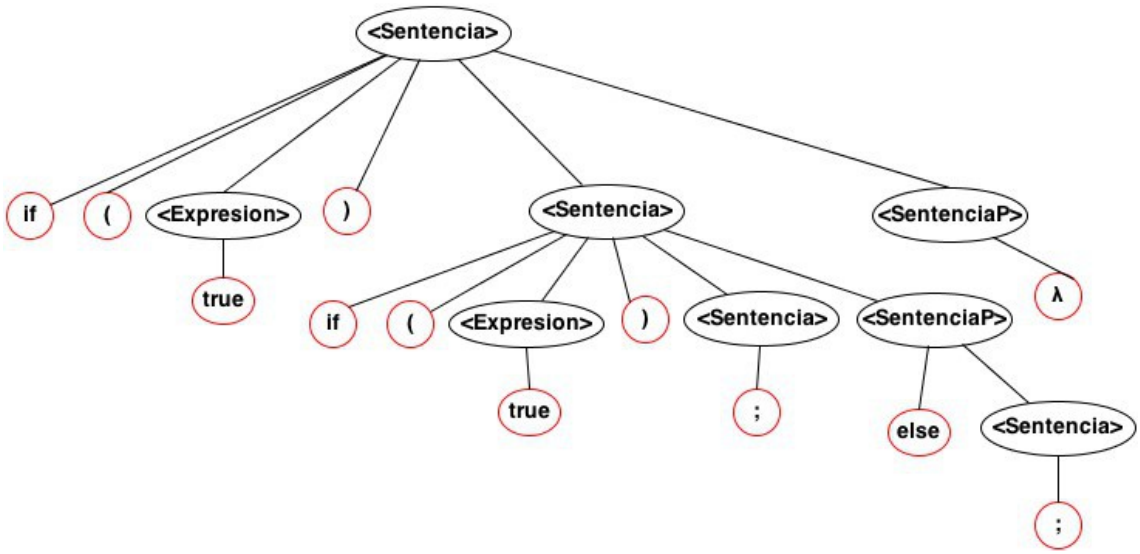
ntencia> → **if (** <Expresion> **)** <Sentencia> <SentenciaP>

<SentenciaP> → **else** <Sentencia> | λ

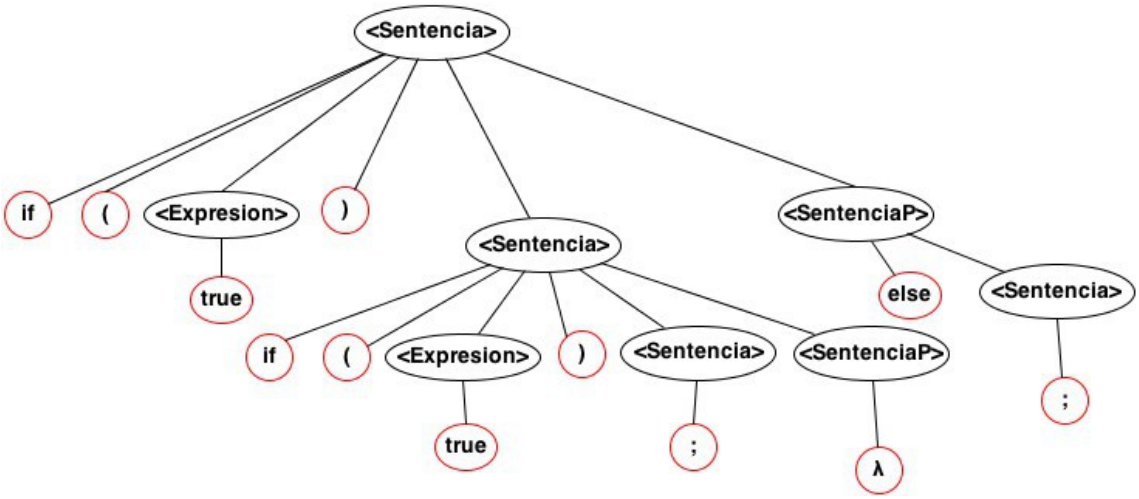
Justificación:

Para la cadena *"if (true) if (true) ; else ;"* que representa una sentencia, es posible obtener dos árboles de derivación distintos. El problema es que no se especifica en la gramática a que "**if**" corresponde el primer "**else**" encontrado.

(1) Primer árbol de derivación.



(2) Segundo árbol de derivación.



Solución de la implementación.

En la implementación se soluciona la ambigüedad de forma tal que siempre se corresponde con el árbol de derivación (1). Esto es que el primer "**else**" siempre se corresponde con el "**if**" más cercano.

Forma de uso.

<PROGRAM_NAME> <IN_FILE> [<OUT_FILE>]

Para realizar el compilado de un programa fuente escrito en lenguaje MiniJava se deben seguir los siguientes pasos:

1- Abrir una consola del sistema operativo.

2- Moverse hasta el directorio que contiene el archivo minijava.jar

3- Ejecutar uno de los siguientes comandos:
   3.1- java -jar minijava.jar input output
   3.2- Java -jar minijava.jar input

<in_file> : es el archivo de entrada correspondiente al programa escrito en lenguaje MiniJava y es el que se desea compilar.

<out_file> : es el archivo de salida que corresponde al código generado como resultado de la compilación del parámetro <in_file> escrito en lenguaje CEIASM (lenguaje definido por la cátedra).

De no especificarse el parámetro opcional <out_file> el código generado por el compilador será almacenado en un archivo por defecto, con la extencion *.ceiasm .

Para ejecutar un programa compilado en la máquina virtual CeIVM, se deben seguir los siguientes pasos:

1- Abrir una consola del sistema operativo.

2- Moverse hasta el directorio que contiene el ejecutable de la máquina virtual CeIVM, por ejemplo: CeIVM.jar

3- Ejecutar el siguiente comando:
   java –jar CeIVM.jar archivo_compilado
   donde el “archivo_compilado” es el archivo resultado de la compilación realizada por el compilador de MiniJava desarrollado.