Universidad de Zaragoza

Procesadores de Lenguajes

Compilador de MiniLeng

Informe sobre el desarrollo del compilador

Fernando Peña Bes (NIA: 756012)

Zaragoza, España

Curso 2019 – 2020

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Resumen

En este informe de documenta el desarrollo de un compilador para el lenguaje MiniLeng durante la asignatura Procesadores de Lenguajes.

MiniLeng es un lenguaje procedural, estructurado y fuertemente tipado. Su sintaxis está inspirada en Pascal.

Para implementar el compilador, se ha utilizado JavaCC, un metacompilador de código abierto para el lenguaje Java. Se ha desarrollado, principalmente, sobre Eclipse 2020-06 con el plugin JavaCC Eclipse Plug-in 1.5.33.

Índice

1. Análisis léxico

El analizador es case-insensitive, no distingue entre mayúsculas y minúsculas.

1.1. Expresión regular utilizada para los identificadores

Según la especificación del lenguaje, los identificadores están compuestos por letras, números y el símbolo '_', no pueden comenzar por número y no pueden terminar en '_'.

Una posible expresión regular que reconoce identificadores es la siguiente:

```
< #DIGITO : [ "0"-"9" ] >
< #LETRA : [ "a"-"z" ] >
< tIDENTIFICADOR :
  (< LETRA > | "_"(< LETRA > | < DIGITO >))(("_")?(< LETRA > | < DIGITO >))* >
```

1.2. Procesamiento de comentarios

En el lenguaje se permiten definir comentarios de una línea (simples) y multilínea.

Los comentarios de una línea se indican con el carácter '`%`', todo lo que haya desde ese carácter hasta el indicador de fin de línea se considerará como comentario. Los comentarios multilínea empiezan y terminan con la secuencia '`%%`'.

Para implementar los comentarios en JavaCC se han utilizado contextos léxicos. Inicialmente el analizador se encuentra en el contexto `DEFAULT`. Se puede indicar el contexto al que pertenece una declaración léxica entre los signos '<' y '>', de forma que sólo se tendrá en cuenta cuando el contexto actual sea ese. Para realizar un cambio de contexto después de reconocer una expresión utiliza el signo '`:`'.

El objetivo es filtrar los comentarios para que no sean enviados al analizador sintáctico. JavaCC define la declaración léxica `SKIP` que permite justamente descartar la entrada reconocida. Teniendo estos aspectos en cuenta, la implementación propuesta para procesar comentarios es la siguiente:

```
SKIP : {  " "| "\r"| "\t"| "\n"
| < "%"~["%"] > : COMENTARIO
| "%%"          : MULTICOMENTARIO}

// Ignorar comentarios en la entrada
<COMENTARIO> SKIP : {
  "\n" : DEFAULT
}

<COMENTARIO> MORE : {
   < ~[] >
}

<MULTICOMENTARIO> SKIP : {
  "%%" : DEFAULT
}

<MULTICOMENTARIO> MORE : {
   < ~[] >
}
```

Las expresiones reconocidas en las declaraciones `MORE` son añadidas como prefijo a la siguiente expresión reconocida. En este caso se ha utilizado la expresión `< ~[] >` dentro de declaraciones `MORE` para descartar la entrada mientras se está dentro de un contexto de comentario.

1.3. Implementación del modo *verbose*

El compilador dispone de una opción *verbose*. Cuando está activada, se muestra al final de la compilación una trabla con el número de ocurrencias de cada token que se han encontrado en el programa. La tabla tiene la siguiente forma:

```
+-------------------------------------+
| Número de ocurrencias de los tokens |
+-------------------------------------+
| Token                          num. |
+-------------------------------------+
| Palabras reservadas                 |
|     PROGRAMA                     1  |
|     PRINCIPIO                    3  |
|     FIN                          3  |
|     SI                           3  |
|     ENT                          3  |
|     SI_NO                        2  |
|     FSI                          3  |
| Caracteres de agrupación            |
|     PARENTESIS_IZQ              27  |
|     PARENTESIS_DER              27  |
| Operadores aritméticos              |
|     SUMA                         2  |
|     RESTA                        1  |
| Operadores lógicos                  |
|     AND                          2  |
|     OR                           1  |
| Tipos de dato                       |
|     BOOLEANO                     1  |
|     CARACTER                     4  |
+-------------------------------------+
```

Para implementarla se ha creado una clase llamada `TablaOcurrencias` en el paquete `lib.lexico`. Esta clase contiene una enumeracion para cada tipo de token (Palabras reservadas, Operadores lógicos, Tipos de dato, Valores...) con los nombres de los tokens que pertenecen al tipo. Cada tipo tiene asociado un vector con un contadores para cada uno de sus tokens, que se van incrementando conforme se reconocen en el código. Para incrementar los contadores, la clase define una función `incrementar` que se llama desde el analizador léxico pasándole el token cuyo contador se quiere incrementar.

La expresión del analizador léxico que reconoce la palabra reservada `PROGRAMA`, realiza la siguiente llamada a la clase:

```
TOKEN : /* PALABRAS RESERVADAS */
{
  < tPROGRAMA : "programa" >
  {
    tabla.incrementar(TablaOcurrencias.Reservadas.tPROGRAMA);
  }
  ...
}
```

1.4. Gestión de errores léxicos

Si mientras se está procesando un programa se encuentra un error léxico, el compilador termina y muestra un error como el siguiente:

```
ERROR LÉXICO (línea ..., columna ...): símbolo no reconocido: '...' (\u...)
```

Si el carácter que ha producido el error no es ASCII, se muestra escapado con su código Unicode.

En JavaCC, los errores léxicos son capturados como excepciones de tipo `Error` durante la ejecución del compilador. Cuando se captura esta excepción, se toma el carácter que ha producido el error junto con su fila y columna en la entrada y se muestran utilizando una función personalizada (definida en la clase `ErrorLexico` de `lib.lexico`).

2. Análisis sintáctico

2.1. Modificaciones sobre la gramática propuesta y decisiones de diseño

En la gramática propuesta, algunas reglas estaban incompletas. A continuación, se detalla como se han completado y cuáles han sido las decisiones de diseño adoptadas.

Parametros formales Contiene la definición de los parámetros de una acción, esta estructura es opcional por lo que se ha utilizado el símbolo '`?`':

```
parametros_formales ::= ( parentesis_izq ( lista_parametros )? parentesis_der )?

lista_parametros ::= parametros ( fin_sentencia parametros )*
```

Lista de sentencias

```
lista_sentencias ::= sentencia ( sentencia )*

sentencia ::= ( leer | escribir | identificacion | seleccion | mientras_que )


leer ::= <tLEER> parentesis_izq lista_asignables parentesis_der fin_sentencia

lista_asignables ::= identificadores

escribir ::= <tESCRIBIR> parentesis_izq lista_escribibles parentesis_der fin_sentencia
lista_escribibles ::= lista_expresiones




seleccion ::= <tSI> expresion <tENT> lista_sentencias ( <tSI_NO> lista_sentencias )* <tFSI>
```

Expresiones

```
lista_expresiones ::= expresion ( sep_variable expresion )*

expresion ::= expresion_simple ( operador_relacional expresion_simple )?
operador_relacional ::= ( <tIGUAL> | <tMENOR> | <tMAYOR> | <tMAI> | <tMEI> | <tNI> )
expresion_simple ::= ( <tMAS> | <tMENOS> )? termino ( operador_aditivo termino )*
operador_aditivo ::= ( <tMAS> | <tMENOS> | <tOR> )
termino ::= factor ( operador_multiplicativo factor )*
operador_multiplicativo ::= ( <tPRODUCTO> | <tDIVISION> | <tMOD> | <tAND> )
```

Los símbolos '`,`', '`;`', '`(`' y '`)`' se han sustituido en la gramática por sus tokens correspondientes: `<tSEP_VARIABLE>`, `<tFIN_SENTENCIA>`, `<tPRENTESIS_IZQ>` y `<tPARENTESIS_DER>`, de esta forma sería posible cambiar los caracteres en un futuro de forma sencilla.

2.2. Gestión de errores y avisos (*warnings*) sintácticos

3. Tabla de símbolos

3.1. Función de hash

3.2. Implementación de la tabla y manejo de colisiones

4. Análisis semántico

4.1. Gestión de errores y avisos (*warnings*) semánticos

4.2. Controles sobre los tipos de datos

4.3. Propagación y manejo de valores constantes en las expresiones

4.4. Comprobaciones sobre parámetros `VAL` y `REF`

No se van a usar parámetros ocultos. Como estamos trabajando con objetos Java, las acciones pueden tener una lista de referencias a los parámetros. Los parámetros se borrarán de la tabla en el momento que la función deje de ser visible y se borre.

5. Generación de código

6. Mejoras introducidas

6.1. Mejoras en el comportamiento del compilador

Errores en el uso del programa

- Si se utiliza una opción inválida o no se especifican correctamente, se muestra el siguiente error:

```
MiniLeng: Opción inválida <error>
```

- Si el fichero a compilar no tiene extensión `.ml`, se muestra el error:

```
MiniLeng: El fichero a compilar tiene que tener extensión .ml
          Fichero introducido: '...'
```

- Si no se encuentra el fichero a compilar, se indica de la siguiente forma:

```
MiniLeng: No se ha encontrado el fichero '...'
```

Al final de la compilación se muestra un recuento de los errores encontrados:

```
Errores léxicos: ...
Errores sintácticos: ...
Veces activado panic mode: ...
```

Y se muestra

```
No se ha podido compilar el programa
```

si ha habido errores durante la compilación, o:

```
Compilado sin errores
```

si la compilación ha sido exitosa.

Si durante la compilación se ha activado el modo pánico, se muestra el siguiente mensaje:

```
Se ha activado el panic mode durante la compilación. Corrige los errores y vuelve a compilar.
```

para indicar al usuario que tiene que hace falta volver a compilar el programa.

6.2. Mejoras en el análisis léxico

La opción tokens muestra por pantalla los nombres de los tokens que se van reconociendo conforme se analiza el programa. Ejemplo:

```
tPROGRAMA
tIDENTIFICADOR (Valor: mi_programa)
tFIN_SENTENCIA
tENTERO
tIDENTIFICADOR (Valor: n)
tFIN_SENTENCIA
...
tFIN
```

6.3. Mejoras en el análisis semántico

- Se permiten acciones anidadas con cualquier nivel de profuncidad. Las declaraciones de acciones anidadas deben entre la zona de declaración de las variables y la palabra 'principio' de la función padre:

```
accion accion1;
  % Declaración de variables de accion1
  accion accion2;
    % Declaración de variables de accion2
    accion accion3;
      % Declaración de variables de accion3
    principio
      % Sentencias de accion3
    fin
  principio
    % Sentencias de accion2
  fin
principio
  % Sentencias de accion1
fin
```

- Se permiten bloques 'seleccion' y 'mientras que' anidados con cualquier nivel de profundidad:

```
SI <condicion> ENT
  % Sentencias
  SI <condicion> ENT
    % Sentencias
  SI_NO
    % Sentencias
  FSI
  % Sentencias
SI_NO
  % Sentencias
FSI
```

```
MQ <condicion>
  % Sentencias
  MQ <condicion>
    % Sentencias
  FMQ
  % Sentencias
FMQ
```

- Cualquier bloque ('principio/fin', 'seleccion' o 'mientras que') debe contener al menos una sentencia.

6.3.1. Panic Mode

El compilador dispone de la opción panic (`-p`), que activa el modo pánico durante la compilación. Se entra en este modo cada vez que se produce un error sintáctico porque se esperaba un punto y coma. El analizador descarta todos los tokens siguientes al error hasta que encuentra un carácter ';' o el fichero se acaba, entonces sale del modo y sigue analizando la entrada.

Cuando se entra en el modo pánico, se informa al usuario con las siguientes líneas:

```
MiniLeng: ERROR SINTÁCTICO (línea ..., columna ...): Token incorrecto: '...'. Se esperaba ';'
PANIC MODE: Iniciado panic mode
```

Y se va mostrando en una línea diferente cada token descartado:

```
> PANIC MODE: Token descartado: '...'
```

Cuando se sale del modo, se muestra:

```
  > PANIC MODE (línea ..., columna ...): Se ha encontrado ';'
PANIC MODE: Terminado panic mode
```

6.4. Mejoras en la generación de código

7. Pruebas realizadas

Anexo A Gramática de MiniLeng implementada

Anexo B Uso del compilador

```
Compilador de MiniLeng -- v2.2 (abril de 2020)
Autor: Fernando Peña Bes (NIA: 756012)

Uso: minilengcompiler [opciones] [fichero]

Opciones:
  -v, --verbose  Mostrar un resumen de los símbolos utilizados en el programa
  -p, --panic  Compila con panic mode
  -t, --tokens   Muestra los tokens que se van reconociendo
  -h, --help  Imprimir ayuda (esta pantalla) y salir
  --version      Imprimir información de la versión y salir


Ejemplos de uso:
    minilengcompiler
```

```
    minilengcompiler -v
    minilengcompiler -v fichero.ml
    minlengcompiler -p -v fichero.ml
```

El fichero con el programa a compilar tiene que tener extensión .ml

Referencias

[1] *Documentación de JavaCC*. URL: https://javacc.github.io/javacc/ (visitado 13-07-2020).
[2] Javier Fabra y José Neira. *Material de la asignatura Procesadores de Lenguajes*.