PROCESADORES DE LENGUAJES

Práctica 4 Creación de un compilador sencillo

Curso 2017/2018

Índice

1. Introducción

En esta práctica aplicaremos todos los conocimientos adquiridos en la asignatura para la creación de un compilador para un lenguaje de programación sencillo cuya especificación se dará más adelante.

Nuestro compilador traducirá programas escritos en nuestro lenguaje de alto nivel al lenguaje de la máquina virtual ROSSI (ver documento `Rossi.pdf` distribuido con el emulador), el cual podrá ejecutarse en un emulador para esa máquina virtual llamado `TheDoc`.

La implementación del compilador se podrá realizar en Python o en C, a elección del grupo.

2. Especificación del compilador

2.1. Descripción informal del lenguaje

Un programa tras la palabra reservada PROGRAMA y un identificador constará de tres partes:

- Una declaración de variables globales.
- Una zona de declaraciones de funciones.
- Una sentencia compuesta que representará el cuerpo del programa principal.

Las partes primera y segunda son opcionales, mientra que la última es obligatoria y será un conjunto de instrucciones que iran entre las palabras INICIO y FIN. Todas las instrucciones terminarán en punto y coma (;).

Los tipos elementales que admite el lenguaje son los enteros, reales y booleanos. Además, el lenguaje cuenta con tipos compuestos vectoriales: se permite la creación de vectores de elementos de cualquier tipo básico del lenguaje. Para representar un vector se emplea la palabra reservada `VECTOR` seguida de un número (que indica el número de elementos[1]) entre corchetes , la palabra reservada `DE` y el tipo base del vector.

Un ejemplo de definiciones variables s sería:

```
  var
x, y: ENTERO;
    lista: VECTOR [10] DE REAL;
```

Después de definir las variables globales, se definen cero o más subprogramas, que pueden ser procedimientos o funciones. Los primeros representan un conjunto de instrucciones que realizan una tarea pero que no devuelven ningún valor, mientras que las funciones devolverán un valor que han calculado.

Un ejemplo de función sería:

```
  FUNCION compara: BOOLEANO;
   INICIO
    SI x<y ENTONCES INICIO
        compara:=CIERTO;
FIN
    SINO INICIO
  compara:=FALSO;
FIN
FIN
```

[1] Los índices del vector van de 0 al tamaño especificado menos uno

En nuestro programa podremos definir comentarios de bloque y estos deben ir encerrados entre llaves ({}).

Definiremos a continuación de forma mas formal la especificación del lenguaje.

2.2. Especificación léxica

A la hora de definir las diferentes categorías léxicas debemos tener en cuenta las siguientes características y restricciones.

1. Los comentarios aparecerán encerrados entre llaves ({ y }) y dentro de ellos no podrá aparecer el carácter {. Los comentarios pueden aparecer en cualquier punto del código fuente.

2. Los espacios en blanco solamente se utilizan para separar los diferentes componentes léxicos. Se utilizarán tabulaciones y saltos de linea simplemente para hacer el código más legible.

3. Los identificadores **id** se adecuarán a la siguiente definición:

$$\begin{array}{ccc} letra & \longrightarrow & [a-zA-Z] \\ digito & \longrightarrow & [0-9] \\ id & \longrightarrow & letra(letra|digito)^* \end{array}$$

   Estos identificadores pueden tener una longitud máxima.

4. El componente léxico **num** puede definirse como sigue:

$$\begin{array}{ccc} digitos & \rightarrow & digito\ digito^* \\ fraccion\_opt & \rightarrow & .\ digitos|\lambda \\ num & -> & digitos\ fraccion\_opt \end{array}$$

5. El conjunto de palabras clave del lenguaje está formado por las siguientes palabras: PROGRAMA, VAR, VECTOR, ENTERO, REAL, BOOLEANO, PROC, FUNCION, INICIO, FIN, SI, ENTONCES, SINO, MIENTRAS, HACER, LEE, ESCRIBE, Y, O, NO, CIERTO y FALSO. Estas palabras reservadas deben estar escritas en mayúscula.

6. Los operadores del lenguaje son los siguientes

   - Operadores relacionales son: =, <>, <, <=, >=, >
   - Opreadores aritméticos los clasificaremos en dos categorias léxicas:
     - **opsuma**: +, −
     - **opmult** ∗, /
   - . El operador de asignacion es :=

7. Junto a las categorías indicadas anteriormente, utilizaremos también en nuestro lenguaje los signos dos puntos (**:**), punto y coma (**;**), coma (**,**), corchete de apertura y de cierre (**[** y **]**) y los parentesis de apertura y cierre (**(** y **)**)

2.3. Especificación sintáctica

La especificación sintáctica de nuestro lenguaje viene dada por la gramática mostrada en el cuadro 1.

⟨Programa⟩	→	**PROGRAMA id ;** ⟨decl_var⟩⟨decl_subprg⟩ ⟨instrucciones⟩ **.**
⟨decl_var⟩	→	**VAR** ⟨lista_id⟩ **:** ⟨tipo⟩ **;** ⟨decl_v⟩ $\|\lambda$
⟨decl_v⟩	→	⟨lista_id⟩ **:** ⟨tipo⟩ **;** ⟨decl_v⟩ $\|\lambda$
⟨lista_id⟩	→	**id** ⟨resto_listaid⟩
⟨resto_listaid⟩	→	**,** ⟨lista_id⟩ $\|\lambda$
⟨Tipo⟩	→	⟨tipo_std⟩ | **VECTOR [num] de** ⟨Tipo⟩
⟨Tipo_std⟩	→	**ENTERO** | **REAL** | **BOOLEANO**
⟨decl_subprg⟩	→	⟨decl_sub⟩ **;** ⟨decl_subprg⟩ | λ
⟨decl_sub⟩	→	**PROC id ;** ⟨instrucciones⟩ | **FUNCION id :** ⟨tipo_std⟩ **;** ⟨instrucciones⟩
⟨instrucciones⟩	→	**INICIO** ⟨lista_inst⟩ **FIN**
⟨lista_inst⟩	→	⟨instrucción⟩ **;** ⟨lista_inst⟩ $\|\lambda$
⟨instrucción⟩	→	**INICIO** ⟨lista_inst⟩**FIN** | ⟨inst_simple⟩ | ⟨inst_e/s⟩
	→	**SI** ⟨expresion⟩ **ENTONCES** ⟨instruccion⟩ **SINO** ⟨instrucción⟩
	→	**MIENTRAS** ⟨expresión⟩ **HACER** ⟨instruccion⟩
⟨Inst_simple⟩	→	**id** ⟨resto_instsimple⟩
⟨resto_instsimple⟩	→	**opasigna** ⟨expresión⟩ | **[** ⟨expr_simple⟩ **] opasigna** ⟨ expresión⟩ $\|\lambda$
⟨variable⟩	→	**id** ⟨resto_var⟩
⟨resto_var⟩	→	**[** ⟨expr_simple⟩**]** | λ
⟨inst_e/s⟩	→	**LEE (id)** | **ESCRIBE (** ⟨expr_simple⟩**)**
⟨expresión⟩	→	⟨expr_simple⟩ **oprel** ⟨expr_simple⟩ | ⟨expr_simple⟩
⟨expr_simple⟩	→	⟨término⟩ ⟨resto_exsimple⟩ | ⟨signo⟩ ⟨término⟩ ⟨resto_exsimple⟩
⟨ resto_exsimple⟩	→	**opsuma** ⟨termino⟩ ⟨resto_exsimple⟩ | **O** ⟨termino⟩ ⟨resto_exsimple⟩ $\|\lambda$
⟨termino⟩	→	⟨factor⟩ ⟨resto_term⟩
⟨resto_term⟩	→	**opmult** ⟨factor⟩ ⟨resto_term⟩ | **Y** ⟨factor⟩ ⟨resto_term⟩ $\|\lambda$
⟨factor⟩	→	⟨variable⟩ | **num** | **(** ⟨expresión⟩ **)** | **NO** ⟨factor⟩ | **CIERTO** | **FALSO**
⟨signo⟩	→	$+\|-$

Cuadro 1: Especificación sintactica de nuestro lenguaje

2.4. Especificación semántica del lenguaje

Para finalizar la especificación completa de nuestro lenguaje, definiremos las restricciones semánticas que deberemos comprobar.

1. No se podrán definir dos objetos distintos con el mismo nombre.
2. No podrá haber identificadores que coincidan con palabras clave del lenguaje.
3. Se producirá una conversión implícita de números enteros en reales, tanto en expresiones como en asignaciones.
4. La representación interna de los valores booleanos será 0 para FALSO y cualquier otro número para CIERTO.
5. No habrá conversión implícita de tipos con el tipo booleano.
6. Los nombres de las funciones pueden aparecer como valor derecho en una expresión, pero no como valor izquierdo, excepto en el cuerpo de su propia función.
7. Los nombres de los procedimientos son instrucciones, por tanto, no pueden aparecer ni en expresiones ni en asignaciones.
8. En la utilización de los vectories, se debe realizar una comprobación dinámica de que los índices están dentro de los rangos definidos.
9. La semántica de la instrucción LEE implica que el argumento solo puede ser una variable de tipo simple (ENTERO o REAL), mientras que la instrucción ESCRIBE permite como argumentos las expresiones ("expresión").
10. En otros casos no especificados en esta lista, el grupo deberá decidir y justificar la restricción aplicada.

2.5. Árboles de sintaxis abstracta

Tras la fase de análisis de nuestro programa tendremos el mismo representado mediante un conjunto de árboles: uno por cada función más uno representando el programa principal.

Los tipos de nodo que se emplearán para representar las sentencias están representados en el cuadro 2. Para las expresiones, emplearemos los nodos del cuadro 3.

Es interesante reflexionar acerca del atributo tipo que hemos incluido en todos los nodos de las expresiones. En principio, podríamos pensar que lo restringido del lenguaje hace que los únicos nodos que en rigor lo necesitarían fueran los de llamadas a función y de acceso a variables y vectores (los operadores aritméticos siempre tienen como resultado un tipo entero y los de comparación uno lógico). Sin embargo, en un compilador más realista, no se podría deducir el tipo de un nodo de manera directa. Por otro lado la uniformidad facilita las comprobaciones de tipos. Por ello, hemos decidido poner el atributo tipo en todos los nodos.

Además, tenemos un tipo de nodo para representar las funciones, el `Función`. Como atributos, tendrá el identificador de la función, sus listas de variables locales y de parámetros y el tipo devuelto, además de un atributo tipo que indica que se trata de una función. Tendrá como hijo la sentencia compuesta que representa su cuerpo.

Finalmente, hemos tomado la decisión de crear árboles incluso para programas que tengan errores. Esto hace necesario crear un nodo especial, al que llamamos `Vacío` y que representa las construcciones mal formadas encontradas en la entrada.

Cuadro 2: Nodos empleados para representar sentencias.

Tipo de nodo	Representa	Atributos	Hijos
`Asignación`	sentencia asignación	—	las expresiones del valor-i y del valor por asignar
`Si`	sentencia condicional	—	la expresión de la condición, las sentencias correspondientes al `entonces` y al `si_no`
`Mientras`	sentencia repetitiva	—	la expresión de la condición, las sentencias correspondientes al cuerpo
`Escribe`	escritura de una expresión	—	la expresión
`Compuesta`	sentencia compuesta	—	las sentencias que la componen

Cuadro 3: Nodos empleados para representar expresiones.

Tipo de nodo	Representa	Atributos	Hijos
`Comparación`	una comparación	la operación, el tipo	los operandos
`Aritmética`	una operación aritmética	la operación, el tipo	los operandos
`Entero`	un entero	el valor, el tipo	–
`real`	un real	el valor, el tipo	–
`Llamada`	una llamada a función	la función, el tipo	–
`Llamada`	una llamada a procedimiento	el procedimiento	–
`AccesoVariable`	acceso a una variable	la variable, el tipo	–
`AccesoVector`	acceso a un vector	el tipo	el vector, la expresión con el índice

3. Implementación

Para llevar a buen puerto la construcción de este compilador se sugiere seguir el siguiente plan de trabajo.

1. Implementar el analizador léxico, obviando algunas tareas del mismo que se añadirán mas tarde, como la incorporación de los identificadores a la tabla de simbolos.

2. Definir casos de prueba para la verificación del correcto funcionamiento del analizador, y probar nuestro analizador léxico.

3. Implementación del analizador sintáctico en su aspecto básico. Solamente se comprobará si la sintaxis es correcta o no intentando detectar el mayor número posible de errores. Para ello debe implementarse los mecanismos de sincronización vistos en clase. Tambien debe implementarse el sistema de gestión e informe de errores sintacticos.

4. Definir los casos de prueba para la verificación del correcto funcionamiento del analizador, y probar nuestro analizador sintactico. Los errores encontrados serán fruto de la implementación del analizador sintactico. ya que llegados a este punto tendremos un analizador lexico correcto.

5. Implementar la gestión de la tabla de simbólos que posteriormente

6. Modificar el analizador léxico y sintactico para que se puedan comprobar las restricciones semánticas establecidas. Al mismo tiempo introduciremos el codigo correspondiente a la creación del AST de nuestro programa.

7. Por último, y partiendo del AST generaremos el codigo correspondiente a nuestro programa.

4. Cuestiones de organización y evaluación

1. El desarrollo de esta práctica se realizará a lo largo del resto del curso.

2. Al finalizar la misma se deberá realizar una memoria en la que se recojan los siguientes apartados.

   - **Introducción**. Realizar una pequeña Introducción a los objetivos y resultados esperados
   - **Descripción formal del lenguaje**
     - Especificación completa de todas las categorías lexicas del lenguaje y descripción del MDD equivalente
     - Comprobación de que la gramática que define el lenguaje es LL(1).
   - **Detalles de implementación** Descripción de los aspectos mas relevantes de la implementación.
   - **Conclusiones**. Opinión personal sobre la práctica, tiempo empleado y sugerencias para mejorar la misma.

3. La memoria, junto al codigo fuente del compilador deberá entregarse antes del dia 21-Diciembre a las 23:55 horas.

4. Tras la finalización de los analizadores lexico y sintactico, el profesor tendrá una breve entrevista con cada uno de los grupos para resolver dudas y comprobar que todo es correcto hasta el momento. La fecha de esta entrevista será el dia 15-Noviembre en el horario de clase.