PROCESADORES DEL LENGUAJE

PRÁCTICA FINAL | VIPER

Analizador del lenguaje

Curso 2024-25

Campus de Colmenarejo

uc3m | Universidad Carlos III de Madrid

TABLA DE CONTENIDOS

1· INTRODUCCIÓN

La práctica final consistirá en la implementación de un compilador capaz de analizar programas de un lenguaje denominado Viper, que contiene expresiones aritmeticológicas, registros, vectores simples, funciones y estructuras de control básicas. También será capaz de comprobar la corrección en términos de los tipos de datos utilizados.

Tras conocer los fundamentos del análisis léxico, sintáctico y semántico, así como la familiarización con las herramientas de construcción de compiladores, la práctica permitirá conocer la implementación completa de un analizador en las fases léxica, sintáctica y semántica, y emplear estos conocimientos para reconocer errores en el uso de tipos e informar al programador.

2· OBJETIVOS

La práctica se divide en las siguientes partes o bloques de contenido a realizar:

I. Generación del analizador léxico, que reconozca los elementos del lenguaje a este nivel.
II. Generación de la gramática, que permitirá reconocer el lenguaje proporcionado y los símbolos terminales y no terminales de la misma mediante el analizador sintáctico creado a partir de la misma.
III. Añadir la capacidad para registrar y tratar variables declaradas por el programador, almacenado el tipo con el que se declaren.
    a. Se deberá crear una estructura de tipo tabla para almacenar los símbolos.
        i. Esta servirá para registrar la información de las variables de tipo básico del programa y, posteriormente, las de tipo vector.
    b. Se deberá crear una estructura de tipo tabla para almacenar los registros.
        i. Esta servirá para almacenar los tipos complejos declarados en el programa, de tipo registro.
IV. Comprobaciones semánticas de todas las expresiones del programa, incluyendo la inicialización de las variables, las expresiones asignadas a variables, operadores aplicados y condiciones de los controles de flujo.
    a. Se harán comprobaciones de tipos y operaciones válidas para distinguir los diferentes tipos del lenguaje incluyendo, cuando sea necesaria, la conversión de tipos compatibles.
V. Y, por último, se tendrá que elegir uno de los dos siguientes apartados para poder obtener la calificación máxima:
    a. Extender las tablas de variables y tipos para registrar las funciones definidas y hacer comprobaciones semánticas en las llamadas a función.
    b. El compilador se recuperará de errores a diferentes niveles (léxico, sintáctico y semántico). Consultar la teoría para más detalles.

3- ESPECIFICACIÓN DEL LENGUAJE DE ENTRADA

El lenguaje Viper es un pequeño lenguaje de programación imperativo fuertemente tipado. El lenguaje trabaja con varios tipos de datos básicos y permite definir tipos compuesto de tipo registro y vectores. Las estructuras de control de flujo básicas que contiene son el salto condicional y el bucle condicional. Permite la declaración y llamada de funciones.

```
type Point:
  float x, y

type Circle:
  Point center
  float radius

Circle[2] balls

# 1st ball
balls[0].center.x = 0b101
balls[0].center.y = 0xFF
balls[0].radius = 1.74
# 2nd ball
balls[1].center.x = 10e-1
balls[1].center.y = -31
balls[1].radius = 3.2

int i = 0
while i < balls.len:
  balls[i].center.x = balls[i].center.x + 1
  balls[i].center.y = balls[i].center.y + 2
  i = i + 1

def bool gte(int a, b):
  return a > b

'''
Result should be
equal to 'N', i.e. no
'''
char result
if gte(balls[0].radius, balls[1].radius):
  result = 'Y'
else:
  result = 'N'
```

Ilustración 1 - Ejemplo de fichero Viper

3.1- Comentarios

Se debe permitir la aparición de comentarios, dentro y fuera de las funciones, que deben ser tratados -e ignorados- por completo en el análisis léxico. Los tipos de comentarios son:

I. De una línea: Comienzan por una almohadilla `#` y terminan con un salto de línea.
II. Múltiple línea: Comienzan por tres comillas simples `'''` y terminan con tres comillas simples `'''`

3.2- Palabras reservadas

Las palabras reservadas del lenguaje tienen una condición especial y, por tanto, no pueden usarse como nombres de variables. Solo pueden usarse en minúsculas y son:

`true`	`false`	`int`	`float`	`char`
`def`	`return`	`type`	`if`	`else`
`and`	`or`	`not`	`bool`	`while`

3.3- Variables, tipos y operaciones

I. Números enteros
   a. Base decimal - `0` `10` `420`
   b. Base binaria - `0b101` `0b110110` `0b0001` (comienzan con `0b`)
   c. Base octal – `0o712` `0o332` `0o1121` (comienzan con `0o`)
   d. Base hexadecimal - `0xED13` `0xAA` `0xFB10` (comienzan con `0x`)
      i. Las letras que componen el número deben ser mayúsculas.

II. Números reales
   a. Notación con punto decimal – `0.1289` `100.001` `1.3140`
      i. Es necesario que el número tenga parte entera y parte decimal.
      ii. Se utiliza `.` como carácter de separación.
   b. Notación científica – `10e-1` `9.87e-2` `5e5`
      i. Se utiliza `e` como carácter de separación.

III. Carácter
   a. Cualquier carácter de la codificación ASCII-extendido.
   b. Delimitado por comillas simples, i.e. `'a'`

IV. Booleanos: Se utilizarán las palabras reservadas `true` y `false`.

V. Vectores: Consultar la sección de vectores.

VI. Registros: Consultar la sección de variables de tipo registro.

Todas las variables deberán ser tipadas en su declaración. A continuación, se detallan los tipos básicos que se manejan en Viper.

Variable	Tipo
Número entero	`int`
Número real	`float`
Carácter	`char`
Booleano	`bool`

Tabla 1 - Tipos básicos de variables

Los tipos de operaciones que se podrán ejecutar son:

I. Aritméticas: Operaciones binarias en notación infija `+` `-` `*` `/` y unarias `+` `-`
II. Booleanas: Conjunción `and` , disyunción `or` y negación `not`
III. Comparación: Igual `==` , mayor `>` , mayor o igual `>=` , menor `<` y menor o igual `<=`

3.4- Sentencias

I. Evaluación de expresiones, de cualquier tipo o combinación válida de operaciones aritméticas, booleanas o de comparación.
II. Declaración/Asignación de variables de los diferentes tipos.

Las sentencias se separan con uno o muchos saltos de línea (al menos uno).

Las declaraciones comienzan por el tipo de la variable seguido de el/los nombre/s de la/s variable/s. Las asignaciones comienzan por un nombre de variable, seguido del signo igual y terminan con una expresión.

- El lenguaje permite declarar una lista de variables separadas por comas. En caso de realizar la asignación a la vez que la declaración, las variables se inicializarán todas con el mismo valor.
- Se permite la asignación en cadena, donde la variable más a la izquierda es asignada primero y se sigue el orden hasta llegar a la de más a la derecha.

```
''' Chained assignments '''
int a, b, c, d = 0
a = c = d = 2 * 3
# a, c y d son iguales a 6
# b es igual a 0
```

Ilustración 2 - Asignación en cadena

Las expresiones pueden ser literales u operaciones matemáticas. No se permite la asignación de valores en mitad de una expresión, es decir, serán incorrectas las sentencias de este tipo:

```
A + 4 * B = 3 - 7 # Error: asignación (B=3) en una expr.
```

Ilustración 3 - Error por asignación dentro de una expresión

Las variables pueden asignarse en el momento de la declaración. En caso de realizar la declaración y asignación por separado, la variable debe declararse antes de asignarse:

```
int a, b
a = 10
b = a * b <= 200
c = '$' # Error: la variable c no ha sido declarada aún

float d, e = 0xFF / (0b101 * (0o702 - 1e-10))
```

Ilustración 4 - Declaración/Asignación

No se permiten las re-declaraciones: cuando se declara una variable, no se podrá volver a declarar ninguna otra variable con el mismo nombre – los nombres de las variables son sensibles a mayúsculas. Esto quiere decir que será legal declarar variables con las mismas letras, pero diferente capitalización (`miVariable`, `MIVARIABLE`, `MiVariable`, etc.)

```
bool var1, VAR1, Var1
float var1 # Error: la re-declaración no está permitida

VAR1 = false # Ok: VAR1=(bool, false)
VAR1 = true # Ok: la reasignación sí está permitida
```

Ilustración 5 - Re-declaración y reasignación

Cuando una variable es declarada sin asignársele un valor inicial (declaración sin asignación), se le asignará el valor por defecto para su tipo de dato, de acuerdo a la siguiente tabla:

Tipo	Valor
int	0
float	0.0
char	\u0000
bool	false

Tabla 2 - Valores por defecto para variables sin inicializar

3.5- Variables de tipo vector

El lenguaje permite declarar variables de tipo vector, de una única dimensión.

```
int[3] vector_3d
vector_3d[0] = 5
vector_3d[1] = 2
vector_3d[2] = 1

int array_length = vector_3d.len # Es igual a 3
vector_3d[1] = vector_3d[2] - 4

type Point:
  float x, y

Point[2] line
line[1].x = 1
line[1].y = 2
'''(0,0) ----> (1,2)'''
```

Ilustración 6 - Variables de tipo vector

I. No se puede hacer declaración con asignación: primero se declara la nueva variable y después se asigna el valor de cada una de las posiciones
    a. El valor inicial de cada uno de los elementos del vector será el valor por defecto del tipo correspondiente.

II. Para su declaración, se seguirá la siguiente sintaxis: `TIPO [ EXPRESION ] id`
   a. `TIPO` ha de ser cualquiera de los tipos de datos básicos o un tipo complejo definido por el usuario.
   b. `EXPRESION` es la longitud fija del vector. Tiene que ser de tipo `int`
   c. `id` es el nombre de la variable declarada
III. Para acceder a una posición concreta del vector: `id [ EXPRESION ]`
   a. `id` es la variable de tipo vector que se está accediendo
   b. `EXPRESION` es el índice que se está accediendo. Tiene que ser de tipo `int` y el valor debería estar dentro de los límites del vector (`0` y `<tamaño> - 1`), pero no puede ser comprobado en tiempo de compilación.
IV. Se puede acceder al tamaño de un vector con su propiedad `len`

3.6- Variables de tipo registro

El lenguaje permite declarar variables de tipo registro, así como los tipos que definen su forma.

```
type Circle:
  float cx, cy, radius
  char color

type Square:
  float side
  char color

Circle c
c.radius = 10.2
c.color = 'R'
int circle_area = 3.14 * c.radius * c.radius

Square s
s.side = 10.0
s.color = 'B'
int square_area = s.side * s.side
```

Ilustración 7 - Variables de tipo registro

I. Para poder declarar una variable de tipo registro, debe haberse definido su tipo con anterioridad.
II. Para declarar un tipo: `type id : \n DECLARACIONES`
   a. `id` es el nombre del tipo que se está declarando.
   b. `DECLARACIONES` son una o muchas declaraciones de variables separadas por saltos de línea. Se permite la multi declaración.
III. No se puede hacer declaración con asignación: primero se declara la nueva variable y después se asigna el valor de cada una de las propiedades.
   a. El valor inicial de cada uno de los elementos del registro será el valor por defecto del tipo correspondiente.
IV. Para declarar un registro: `id id`
   a. El primer `id` corresponde al nombre del tipo declarado previamente.

b. El segundo `id` corresponde al nombre de la variable que se está declarando.

V. Para acceder a una propiedad concreta del registro: `id . id`
   a. El primer `id` corresponde al nombre del registro.
   b. El segundo `id` corresponde al nombre de la propiedad.

VI. Una propiedad de un registro podrá ser a su vez un tipo complejo.

```
type String:
  char[64] _

type Person:
  String name, last_name
  int age
  float height

Person pepe
pepe.name._[0] = 'P'
pepe.name._[1] = pepe.name._[3] = 'e'
pepe.name._[2] = 'p'
pepe.last_name._[0] = 'G'
pepe.last_name._[1] = pepe.name._[5] = 'a'
pepe.last_name._[2] = 'r'
pepe.last_name._[3] = 'c'
pepe.last_name._[4] = 'i'
pepe.age = 32
pepe.height = 1.83
```

Ilustración 8 - Registros con propiedades de tipo registro

3.7- Reglas de tipos

I. Movimiento de datos: Se utiliza un dato previamente definido, hay que comprobar si está permitido usarlo en ese punto.
II. Combinación de datos: Varios datos son combinados para producir un resultado.

Ambos niveles están relacionados: a veces es necesario resolver una operación de combinación de datos para saber si el tipo de dato que se va a producir es válido para una aplicación posterior.

```
int myVariable = 10
float myFloat = 12.12

if myVariable:
  # ...
''' [movimento]
¿se puede utilizar myVariable de tipo int como condición?'''

myVariable + myFloat
''' [combinación]
¿cuál es el tipo resultante de la expresión?'''
```

Ilustración 9 - Categorías en las reglas de tipado

Para ambas categorías, debemos asegurarnos que el tipo de origen y del de destino son iguales, o bien que se puede utilizar una conversión del tipo de origen al tipo de destino sin pérdida de datos: algunos tipos pueden transformarse automáticamente, según la siguiente tabla de conversión.

Origen	Destino	Transformación
`char`	`int`	Se toma el valor numérico del carácter (0-255)[1]
`int`	`float`	Pasar de 32 bits que representan un número enero en CA_2 a 32 bits que representan un real en IEEE754

Tabla 3 – Conversión legal automática de tipos

```
float f1, f2
int i
bool b

f1 = 7.5     # Ok: float -> float
f2 = 0b11    # Ok: int -> float
i = 'a'      # Ok: char -> int
b = true     # Ok: bool -> bool

i = 7.5      # Error: float -> int
b = 7        # Error: int -> bool
b = f1       # Error: float -> bool
```

Ilustración 10 - Asignaciones con diferente tipo válidas y no válidas

Por otro lado, todas las operaciones que combinan/manejan dos o más valores deben asegurar que sus tipos son similares, o al menos que se puede realizar la conversión de uno de ellos al tipo del otro sin pérdida de datos.

Respecto a las operaciones y expresiones, cada operador requiere uno o varios tipos de dato concretos a la entrada, y devuelve así mismo un tipo específico de dato a la salida. En la siguiente tabla se especifican los tipos de datos compatibles con cada operador (sin tener en cuenta conversiones), y la salida correspondiente para cada tipo de entrada:

Operador	Tipo origen	Tipo destino
MÁS (+), MENOS (-)	`int`	`int`
	`float`	`float`
	`char`	`char`
POR (*), ENTRE (/)	`int`	`int`
	`float`	`float`
MAYOR (>), MENOR (<), MAYOR O IGUAL (>=), MENOR O IGUAL (<=)	`int`	`bool`
	`float`	`bool`
	`char`	`bool`

[1] https://docs.python.org/es/3/library/functions.html#ord

Operador	Tipo origen	Tipo destino
IGUAL (==)	`int`	`bool`
	`float`	`bool`
	`char`	`bool`
	`bool`	`bool`
AND (and), OR (or), NOT (not)	`bool`	`bool`

Tabla 4 – Operadores y tipos de datos permitidos

No obstante, cuando una de las entradas de un operador binario tiene un tipo A y la otra entrada un tipo B, será necesario convertir una de ellas al tipo de la otra: el dato con el tipo más restrictivo se transforma al tipo más general:

I. `char` puede convertirse a `int` o `float`
II. `int` puede convertirse a `float`

No se permite la conversión del tipo `bool` a ningún otro tipo. A continuación, se muestran varios ejemplos donde se combinan todas las reglas referentes al tipado:

```
5.1 * 41.0          # Ok: real * real -> real
char c = 'c' + 'c'  # Ok: char + char -> char
int i = 1 + 'c'     # Ok: int + char -> int

'''
Ok: 1) 'c' * 8 -> char * int -> int
    2) 7.5 + (1) -> float + int -> float
'''
float f1 = 7.5 + 'c' * 8

'''
Error: 1) 'd' -> char -> int (automatic conversion)
       2) (1) < 8 -> int < int -> bool
       3) 5.0 + (2) -> float * bool -> TYPE ERROR
'''
float err1 = 5.0 * ('d' < 8)

'''
Error: "if" require boolean, no se puede convertir
directamente de real a bool
'''
if 5 / 7:
  err1 = 0.0
```

Ilustración 11 - Ejemplos de reglas referentes al tipado

3.8- Control de flujo

En cuanto al control de flujo, se definen dos estructuras de control: salto condicional y bucle condicional.

3.8.1- Salto condicional

El salto condicional es una sentencia que evalúa una expresión booleana – llamada condición – y, dependiendo de si el resultado es verdadero o falso, salta a un punto más avanzado del programa o sigue ejecutando las sentencias adyacentes.

Una de las posibles formas de definir la construcción del salto condicional podría ser:

```
if EXPRESION : \n BLOQUE_SENTENCIAS
```

- Si la expresión de la condición evalúa verdadero, se ejecutará el bloque de sentencias a continuación de los dos puntos y el salto de línea y después, se continuará el programa con normalidad.

```
if EXPRESION : \n BLOQUE_SENTENCIAS \n else BLOQUE_SENTENCIAS
```

- Si la condición evalúa verdadero, se ejecutará el primer bloque de sentencias y después se saltará a la sentencia siguiente al salto condicional.
- Si la condición evalúa falso, se ejecutará el segundo bloque de sentencias y luego se continuará con normalidad.

```
if f2 == 3 || b1 && b2 || 10 - 5 * i1 >= 0xFF - 1e-1:
  f2 = f2 - 3
else:
  f2 = 10 - f1 * f1
```

Ilustración 12 - Ejemplo de código con control de flujo de salto condicional

3.8.1- Bucle condicional

El bucle condicional es una sentencia que evalúa una expresión booleana – llamada condición – y, dependiendo de si el resultado es verdadero o falso, ejecuta el cuerpo del bucle o bien salta al final de éste.

Una de las posibles formas de definir la construcción del bucle condicional podría ser:

```
while EXPRESION : \n BLOQUE_SENTENCIAS
```

```
int i, var = 0

while i < 10:
  var = (var + 1) * 2
  i = i + 1
```

Ilustración 13 - Ejemplo de código con control de flujo de bucle condicional

3.9- Funciones

Una función es un bloque de sentencias que, dados unos parámetros de entrada, realizan una serie de cálculos para obtener un único valor al que se denomina retorno.

Las funciones tienen un nombre que permite que sean referenciadas desde cualquier punto de un programa, al igual que ocurre con las variables.

Para declarar una función es necesario especificar:

- El nombre de la función
- El nombre de sus valores de entrada o argumentos
- El tipo de retorno
- Las sentencias que forman parte del cuerpo de la función
- La sentencia de retorno

En Viper los argumentos no son obligatorios: una función puede recibir cero valores de entrada. Sin embargo, el cuerpo de una función debe contener, al menos, la sentencia de retorno:

```
def TIPO id ( LISTA_ARGUMENTOS ) : \n BLOQUE_SENTENCIAS return SENTENCIA
```

La lista de argumentos será una serie de identificadores separados por comas, indicando el tipo de los mismos. Como se comentaba previamente, es válida la lista de argumentos vacía.

Una función es invocada cuando se escribe su nombre, seguida de una lista de expresiones entre paréntesis que coincide en número y tipo con sus argumentos. Esto implica que puede existir la sobrecarga de funciones (mismo nombre, distinto número de argumentos o tipo de los mismos).

```
id ( LISTA_EXPRESIONES )
```

El resultado será un valor con el tipo de dato de retorno de la función. Hay que recordar que las expresiones que se pasan en la llamada de la función pueden ser operaciones o combinaciones de las mismas, variables o directamente valores del tipo esperado.

```
def int mod(int a, b):
  while a >= b:
    a = a - b
  return a

def int greatest_common_divisor(int a, b):
  int temp
  while not b == 0:
    temp = b
    b = mod(a, b)
    a = temp
  return a

int result = greatest_common_divisor(132, 0xFF)
```

Ilustración 14 - Ejemplo de código con definición y llamada de funciones

Recordamos que la extensión las tablas de variables y tipos para registrar las funciones definidas y hacer comprobaciones semánticas en las llamadas a función es una modificación avanzada.

También podrá usarse los tipos complejos definidos por el desarrollador (variables de tipo registro) como tipo de los argumentos de entrada o del retorno de la función.

```
type String:
  char[64] _

type User:
  int user_id
  String nickname

int global_id = 0

def User new_user(String nickname):
  User user
  u.user_id = global_id
  global_id = global_id + 1
  u.nickname = nickname
  return user

String nickname
nickname._[0] = 'V'
nickname._[1] = 'i'
nickname._[2] = 'p'
nickname._[3] = 'e'
nickname._[4] = 'r'
new_user(nickname)
```

Ilustración 15 - Ejemplo de código con función usando tipos complejos

4. RECUPERACIÓN DE ERRORES

La recuperación de errores trata de informar con claridad y exactitud (tipo de error, línea y columna en que se producen) de los errores que ocurren en los diferentes niveles de análisis. Debe ofrecer una recuperación rápida que permita continuar con el análisis, sin retrasar el procesamiento del programa sin errores.

Algunos ejemplos de error serían:

I. Léxico: Escribir mal un identificador, caracteres no reconocidos, etc.
II. Sintáctico: Falta un salto de línea entre sentencias.
III. Semántico: Multiplicar una variable booleana por una de tipo entero.

El objetivo de esta modificación avanzada consiste en detectar todos los errores, lo antes posible, evitando detectar errores repetidos o generar falsos positivos. Se deberá consultar la teoría (Tema 4 – Recuperación de errores) para más detalle.

Será importante acompañar una detallada descripción de las estrategias implementadas en la memoria.

5· ENTREGA PARCIAL

En una primera entrega se deberá completar la construcción del analizador léxico del lenguaje y la especificación de la gramática. El objetivo principal de esta entrega es conseguir una gramática sin conflictos ni ambigüedades, de cara a completar la entrega final sin acarrear errores previos.

Cada pareja deberá entregar todo el contenido de su práctica en un único archivo comprimido en formato zip. El nombre del comprimido debe seguir el siguiente formato `AP1_AP2_PL_P2.zip`

- AP1 y AP2 son los primeros apellidos de los integrantes en orden alfabético.
- Dentro del fichero zip, habrá una única carpeta con el mismo nombre.

Se realizarán pruebas extensivas tanto del analizador léxico como del sintáctico, por lo que:

I. Se generará un fichero con el nombre del fichero de entrada, pero extensión ".token", que incluirá una línea por cada token reconocido donde aparezca el tipo y el valor del token separados por un espacio en blanco.
II. El análisis sintáctico deberá ser correcto, lo que significa que no deberá aparecer ningún conflicto y/o error sintáctico.

```
# Fichero entrada - example.vip
int a = 28
float b = 24.9

# Ejecución
> python main.py example.vip
>> Generating LALR tables
>> ... # Sin conflictos, ni errores

# Fichero salida - example.token
INT int
ID a
EQUAL =
INT_VALUE 28
NEW_LINES \n
FLOAT float
ID b
EQUAL =
FLOAT_VALUE 24.9
NEW_LINES \n
```

Ilustración 16 - Ejemplo de ejecución y salida del analizador léxico

Sea adjuntara una breve, pero completa memoria que profundice en el razonamiento y proceso de toma de decisiones de los autores. Deberá servir como punto de partida para la memoria de la entrega final, donde la sección de análisis léxico no debería tener mucha variación. El aspecto más importante de esta memoria es explicar, detalladamente, el diseño de la gramática: cuál es la estructura principal de lenguaje, que se permite y qué no y como se ha definido esto en el diseño, cómo funcionan las expresiones, etc.

Aunque esta entrega será calificada, se podrán realizar cambios en el proyecto después de obtener la retroalimentación de cara a la entrega final.

6· ENTREGA FINAL

{El documento será actualizado más adelante}