Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC3102 Teoría de la Computación

Tarea N°2

Interpretes

Profesor: Gonzalo Navarro
Auxiliar: Rodrigo Fuentes
Ayudantes: Javiera Alegría
Gabriel Chaperón
Matías Rojas
Alumnos: Joaquín Cruz
Diego Ortego
Valeria Valdés
Fecha de Realización: December 16, 2018
Fecha de Realización: 16 de Diciembre de 2018

1 Introducción

Esta tarea consiste en generar un lexer y un parser capaces de reconocer y ejecutar instrucciones escritas en un lenguaje de programación básico descrito en una pequeña gramática.

Para esto se ocupo la librería de código para `Python` llamada `ply`.

El lenguaje debe ser capaz de trabajar con números enteros, guardar variables, aceptar operaciones aritméticas, recibir input del usuario a través de la terminal, imprimir output a la misma terminal, calcular condiciones, y calcular adecuadamente instrucciones condicionales y loops.

La labor del lexer es transformar el texto del programa que se quiere ejecutar a `tokens`, reconocidos mediante expresiones regulares, que el parser luego puede reconocer.

El parser, luego, recoge los `tokens` y ocupa una gramática dependiente del contexto para construir un árbol de instrucciones que deben representar inequívocamente al programa.

Finalmente una función programada en python recoge dicho árbol y lo ejecuta.

Las dependencias e instrucciones de compilación del programa se encuentran en el archivo `README.md`, y además hay algunos ejemplos de uso en la sección 3.

2 Descripción del Programa

2.1 Formato de Ejecución

El programa principal recibe un input, el cual corresponde a la dirección del archivo donde está el código que se quiere ejecutar, relativa a su ubicación.

Por ejemplo `./code.txt`, `./src/prog.out` o `./../../Downloads/exec.ris`.

El programa empieza imprimiendo el texto:

```
Enter filename to execute (or press enter to end):
>>
```

El marcador `>>` indica que se esta pidiendo un input para el programa principal de `Python`.

Luego de que un ejecutable ha sido especificado, las query que el ejecutable haga al usuario se pueden distinguir porque ocupan el marcador `#>`.

Cada vez que un código ejecutable termina, el programa principal vuelve a preguntar por otro que ejecutar.

Para terminar la ejecución basta con ingresar un input vacio al programa principal, como especifica su mensaje arriba.

En caso de que el ejecutable que se está buscando no exista, el programa se reinicia, levantando el siguiente error:

```
$ python program.py
Enter filename to execute (or press enter to end):
>> test_3
[Errno 2] No such file or directory: 'test_3'
Enter filename to execute (or press enter to end):
>>
```

2.2 Generación del Lexer

Los `token` para cada símbolo del lenguaje fueron los siguientes:

```
< Num >     ::= INT
< Var >     ::= NAME
< + >       ::= PLUS
< - >       ::= MINUS
< * >       ::= MULTIPLY
< / >       ::= DIVIDE
< == >      ::= EQUALS
< != >      ::= NOTEQ
< <= >      ::= LOWEREQ
< >= >      ::= GREATEREQ
< < >       ::= LOWER
< > >       ::= GREATER
< = >       ::= ASSIGN
< read() >  ::= READ
< print >   ::= PRINT
< ; >       ::= NEXTINST
< ( >       ::= LEFTBRACKET
< ) >       ::= RIGHTBRACKET
< { >       ::= WLEFTBRACKET
< } >       ::= WRIGHTBRACKET
< while >   ::= WHILE
< do >      ::= DO
< if >      ::= IF
< then >    ::= THEN
< else >    ::= ELSE
```

A partir de estos, se crean expresiones regulares que el lexer debe utilizar para identificarlas. Estas deben ir en el código según lo que el lexer debe buscar primero, por ejemplo, la expresión regular para el `==` debe ir antes que la del `=` pues sino para el lexer el `==` serán dos `=` consecutivos.

Por otro lado, las expresiones regulares complejas fueron las palabras reservadas del lenguaje (`while`, `do`, `then`, etc...) y los símbolos que se pueden construir a partir de la repetición de algunos caracteres (valores numéricos y variables). Para que el lexer distinga a las palabras reservadas del lenguaje se hicieron las funciones que le permiten interpretarlas. Además, se agregaron funciones que le permiten guardar los valores de números y nombres de variables, por lo cual el parser podrá luego usarlos

fácilmente. Finalmente, se permitió al lexer ignorar los espacios, tabulaciones y saltos del linea, permitiendo así que los programas del lenguaje no tengan restricciones sintácticas con esos caracteres.

2.3 Generación del Parser

La gramática original fue modificada, quedando la siguiente:

```
< S >      ::= < Code > |
               < Inst >
< Code >   ::= < Inst > < Inst > |
               < Code > < Inst >
< Inst >   ::= < If >     |
               < While >  |
               < Assign > ; |
               < Read > ;  |
               < Print > ; |
               { < Inst > } |
               { < Code > }
< Assign > ::= < Var > = < Exp >
< Read >   ::= < Var > = read()
< Print >  ::= print(< Exp >)
< If >     ::= if < Exp > then < Inst > else < Inst > |
               if < Exp > then < Inst >
< While >  ::= while < Exp > do < Inst >
< Var >    ::= < Let > | < Var > < Let > | < Var > < Dig >
< Let >    ::= a | b | ... | z | A | ... | Z | _
< Dig >    ::= 0 | 1 | 2 | ... | 9
< Num >    ::= < Dig > < Num > | < Dig >
< Exp >    ::= < Exp > < A > < Exp > |
               < Exp > < C > < Exp > |
               (< Exp >)           |
               - < Exp >           |
               < Num >             |
               < Var >
< A >  ::= + | - | * | /
< C >  ::= == | > | < | != | >= | <=
```

Se añadió el no terminal `<Code>` porque se necesitaba una forma de reunir instrucciones de a dos, que `Python` pudiera luego ejecutar fácilmente en orden.

Se uso específicamente además que `<Code>` e `<Inst>` sean transformables a `<S>` para forzar usando el ordenamiento del programa y las reglas de precedencia que la raíz del árbol de instrucciones que se genera sea, necesariamente, `<S>`, así `Python` esta

preparado para recibir un árbol con un `<Code>` o `<Inst>` como raíz al empezar la ejecución.

Las reglas de precedencia de los símbolos del lenguaje, en orden de menor precedencia a mayor precedencia son:

```
if, then, while, do
else
{, }
< Assign >
=, <, >, !=, <=, >=
+, -
*, /
(, )
```

Esto asegura que `ply` no encuentre ningún error tipo `shift/reduce`.

Por ejemplo, el keyword `else` tiene preferencia por sobre `if` y `then`, para que así el parser sepa encontrar una expresión que incluye el `else` opcional antes de separarlo accidentalmente de sus `if` y `then`.

2.4 Ejecución del Árbol

La forma en que el árbol del programa se ejecuta es desde la raíz hasta las hojas, donde la función de `Python` que recorre el árbol se va llamando a si misma recursivamente.

Si la función encuentra una rama de tipo `<Inst>`, ejecuta su contenido inmediatamente; si encuentra una rama `<Code>` ejecuta la primera instrucción que contiene y luego la segunda.

Las expresiones y condiciones se calculan usando los operadores de `Python`.

Las variables que el lenguaje va guardando se asignan según su nombre a un diccionario de `Python` que se vacía entre ejecuciones y permite llamar los valores en tiempo de ejecución.

En el caso de que el diccionario no contenga una variable, levanta una excepción que reinicia el programa principal e imprime un mensaje especial, como el que se muestra abajo:

```
$ python program.py
Enter filename to execute (or press enter to end):
>> test_4
[Custom Error 1] Tried to access non-existent variable 'a'
Enter filename to execute (or press enter to end):
>>
```

3 Ejemplos de Uso

La ejecución del programa ubicado en el archivo *test_2*:

```
n = read();
f1 = 1;
if (n <= 1) then print(n);
else
    { f2 = 1;
      while (n > 2)
        do
        { f = f1+f2;
          f1 = f2;
          f2 = f;
          n = n-1;
        }
      print (f2);
    }
```

Para calcular Fibonacci de 20:

```
Enter filename to execute (or press enter to end):
>> test_2
#> 20
```

Entrega:

```
6765
Enter filename to execute (or press enter to end):
>>
```

Se ejecuta el archivo *test_3* que calcula el factorial de un número:

```
n = read();
r = 1;
while(n > 0)
  do{
  r = r * n;
  n = n - 1;
  }
  print(r);
```

Se calcula el factorial de 12:

```
Enter filename to execute (or press enter to end):
>> test_3
#> 12
479001600
Enter filename to execute (or press enter to end):
>>
```