CIENCIA DE LA COMPUTACION

COMPILADORES

MANUAL DE REFERENCIA

LENGUAJE PROPUESTO: GRAF

ALUMNO:
Alexander Pinto De la Gala

Mayo de 2020

“El alumno declara haber realizado el presente trabajo de acuerdo a las normas de la Universidad Católica San Pablo”

_____

MANUAL DE REFERENCIA

LENGUAJE GRAF

1. Introducción

Este manual de referencia describe el lenguaje de programación Graf.

1.1 Notación

Las descripciones de análisis léxico y sintáctico utilizan una notación similar a BNF:

name: *lc_letter (lc_letter | "_")**

lc_letter: *"a"…"z"*

La primera línea indica que un name (nombre) inicia con una letra y puede seguir con n letras con sub-guiones.

Una lc_letter (letra) puedes ser un simple carácter de la 'a' a la 'z'.

Los operadores :

- Asterisco * : indica 0 a más
- Suma + : indica uno o más
- *[phrase]* : phrase es opcional
- <…> : información adicional

2. Análisis Léxico

2.1 Estructura de línea

2.1.1 Líneas lógicas

El final de una línea lógica es representado por el token *';'*.

2.1.2 Líneas físicas

El final de una línea física está dado por el caracter *(;)*.

2.1.3 Comentario

Un comentario inicia con el caracter hash *(#)* y termina con la línea física

2.1.4 Unión de líneas

Esta versión no incluye esta característica.

2.1.5 Líneas en blanco

Las líneas en blanco son ignoradas.

2.1.6 Agrupación de sentencias

Las sentencias son agrupadas con el uso de los caracteres *'{'* para iniciar el agrupamiento y *'}'* para terminar el agrupamiento. Se utiliza los mismos tokens para estos caracteres.

2.1.7 Espacio en blanco entre tokens

Los espacios en blanco son utilizados para separar tokens, de manera que no exista confusión entre ellos.

2.2 Identificadores y keywords.

identifier: *(letter|"_")(letter|digit|"_")**

letter: *lowercase | uppercase*

lowercase: *"a" … "z"*

uppercase: *"A"…"Z"*

digit: *"0"…"9"*

Identificadores son ilimitados en longitud. Case sensitive.

2.2.1 Keywords

Los siguientes son palabras reservadas por el lenguaje y no puede ser usados en combinación con identificadores ordinarios:

and or not if then else for in node edge

2.3 Literales

Literales son notaciones para valores constantes de tipos incluidos.

2.3.1 Literales de cadena

Los literales deben estar encerrados entre dobles comillas. *(")*.

stringliteral"cadena"

Se toman en cuenta secuencias de escape de ASCII

\n *Fin de línea*

2.3.2 Literales numéricos

En esta versión hay un literal numérico: plain integer.

integer *: nonzerodigit digit* | "0"*

nonzerodigit *: "1"…"9"*

2.3.3 Literales de objetos

Se incluyen dos literales correspondiente a la abstracción de elementos básicos de una estructura de datos e tipo grafo.

node : *“node “ (“integer” | “string”) “ identifier*

El literal *node* construye un nodo de un grafo en donde su atributo único es del tipo de literal elegido.

edge: “edge” *(“integer” | “string”) “ identifier*

El literal *edge* construye una arista de un grafo en donde su atributo único es del tipo de literal elegido.

2.4 Operadores

Los siguientes tokens son operadores:

*+ - * /*

< > <= >= == ¡=

¡< ¡> ¡ << >>

2.5 Delimitadores

Los siguientes tokens son utilizados como delimitadores en la gramática:

( ) [ ]

3. Expresiones

En adelante la notación BNF extendida será usada para describir la sintaxis, no el análisis léxico.

3.1 Átomos

Átomos son los elementos más básicos de las expresiones. Los átomos más simples son identificadores o literales. Pueden estar encerrados en paréntesis o corchetes.

átomo: identificador | literal

3.1.1 Identificador (Nombres)

Un identificador ocurre cuando un átomo hace referencia al nombre de un binding local o global.

Cuando el nombre es ligado a un objeto, la evaluación de ese átomo conduce hacia el objeto.

3.1.2 Literal

Graf soporta literal de cadena y numérico entero.

literal: stringliteral | integer

La evaluación de un literal conduce a un objeto del tipo dado (*string, integer)* con el valor dado.

Todos los literales corresponden a tipos de datos inmutables, y por lo tanto la identidad del objeto es menos importante que su valor.

3.1.3 Lista

Una lista muestra una serie de expresiones encerradas en corchetes.

list : *"[" expresión "]"*

expresión: *literal | ("," literal) **

Una lista conduce a una lista de objetos. Sus contenidos son especificados al proveer una lista de expresiones, separadas por coma.

3.2 Operadores aritméticos unarios.

Todos los operadores unarios tienen la misma prioridad.

u_expr: *"-" u_expr | "+" u_expr*

El operador unario *"-"* produce el negativo de su argumento numérico.

El operador unario "+" produce ningún cambio a su argumento numérico.

3.3 Operadores aritméticos binarios.

Las operaciones aritméticas binarias tienen niveles convencionales de prioridad.

Existen dos niveles de prioridad:

m_expr: *u_expr | m_expr "*" u_expr | m_expr "/" u_expr*

a_expr: *m_expr | a_expr "+" m_expr | a_expr "-" m_expr*

El operador * de multiplicación produce el producto de sus argumentos. Ambos argumentos deben ser números.

El operador / de división produce el cociente de sus argumentos. El resultado de la división tendrá la función "*floor*" debido al tipo *integer* soportado. Ambos argumentos deben ser números.

El operador *+* de adición produce la suma de sus argumentos. Ambos argumentos deben ser números.

El operador – de sustracción produce la resta de sus argumentos. Ambos argumentos deben ser números.

3.4 Operadores de Objetos

Los operadores de objetos permiten operaciones con la abstracción de la estructura de grafos.

asignación: *ident_nodo1"<<" ident_edge ">>" ident_nodo2*

La arista *ident_edge* es asignada a un nodo 1 y a un nodo 2. Un edge tiene asignados dos nodos como máximo.

Un nodo puede tener asignados varios *edges*.

acceso a edges: *target "=" "!"ident_nodo*

El operador *!* permite el acceso a los *edges* de un nodo esto devuelve en una lista los edges resultantes. El uso de este operador requiere de una asignación.

acceso a nodos: *"¡<" ident_edge | "!>" ident_edge*

Los operadores *!<* y *!>* permiten el acceso a los nodos izquierdo y derecho de un *edge* respectivamente.

3.5 Comparaciones.

Todas las comparaciones tienen la misma prioridad.

comparison: *or_expr (comp_operator or_expr)**

comp_operator: *"<" | ">" | "==" | "<=" | ">=" | !=*

Las comparaciones producen valores enteros: 1 para verdadero, 0 para falso.

Los operadores *<, >, ==, <=, >= y ¡=* comparan valores de dos objetos. Los objetos necesitan ser del mismo tipo para su comparación.

La comparación de los objetos del mismo tipo depende de:

- Los números son comparados aritméticamente.
- Las cadenas son comparadas lexicográficamente, usando equivalentes numéricos.

3.6 Operaciones booleanas

El operador *not* produce 1 si el argumento es falso y 0 si es verdadero.

4. Sentencias simples

4.1 Sentencias de Expresión

Las sentencias de expresión son utilizadas para calcular y escribir un valor.

expresión_stmt: *expresión_list*

4.2 Sentencias de asignación

Las sentencias de asignación son usadas para (re)ligar nombres a valores.

assignment_stmt : (target "=")+ target

4.3 La sentencia *print*

print_stmt : "print" expresión

La sentencia print evalua la expresión y escribe el objeto resultante en una salida.

5. Sentencias compuestas

Las sentencias compuestas contienen grupos de otras sentencias, estas afectan o controlan la ejecución de las otras sentencias de alguna manera. En general las sentencias compuestas contienen múltiples líneas.

5.1 La sentencia *if*

La sentencia *if* es usada como condicional de ejecución:

if_stmt: *"if" expresión "{" suite "}"*
*("else if" expression "{" suite "}")**
["else" "{" suite "}"

Selecciona exactamente una de las *suites* para evaluar las expresiones una por una hasta que una resulte verdadera entonces esa *suite* es ejecutada (y ninguna otra parte de la sentencia *if* es ejecutada).

Si todas las expresiones son falsas la *suite* de la sentencia *else* ejecutada si está presente.

5.2 La sentencia *for*

La sentencia *for* es usada para iterar sobre los elementos de una secuencia:

for_stmt: *"for" target "in" expression_list"*
"{" suite "}"

La *expression list* es evaluda una vez, debería representar una lista. La *suite* entonces es ejecutada una vez por cada item en la secuencia, en el orden de índices ascendientes. Cada item a la vez es asignado al *target* usando reglas de asignación, y luego la *suite* es ejecutada.

Cuando los ítems han sido agotados (lo que significa que la secuencia está vacía) el bucle termina.