Instituto Tecnológico de Costa Rica

Ingeniería en Computación

Compiladores e Intérpretes

Tarea 1

Profesor: Ing. Allan Rodriguez Davila

Estudiantes: Carnet:

Jordano Escalante 2018161994

Fredd Badilla V 2022012800

Primer Semestre

Fecha: 13 de Marzo

Año:2024



**Descripción del problema**

Se requiere desarrollar una Gramática BNF (considerando únicamente aspectos léxicos y sintácticos) para un lenguaje imperativo con las siguientes características:

1. Permitir la creación de funciones, y dentro de ellas, estructuras de control, bloques de código (los bloques finalizan e inician con guión bajo “\_”) y sentencias de código.
2. Manejar los tipos, sus variables y expresiones, enteros, flotantes, booleanas, caracteres, cadenas de caracteres (string) y arreglo estático unidimensional.
3. Permitir sentencias para creación de variables (no permiten guión bajo), creación y asignación de expresiones y asignación de expresiones a variables, y algunos casos, sólo expresiones sin asignación.
4. Se permite crear arreglos de tipo entero o char. Además, se permite obtener y modificar sus elementos, y ser utilizados en expresiones.
5. Se permite la creación de variables globales y locales. Se debe indicar la palabra reservada para identificarlas (glob y loc). Las variables globales podrán crearse dentro o fuera de las funciones y las locales dentro de funciones.
6. En la declaración (variables, funciones y sus parámetros), return con expresión, case del switch, glob y loc existirá un separador entre palabra reservada, tipo y/o id que será “:”. Un programa escrito en este lenguaje podría escribirse sin espacios.
7. Las expresiones permiten combinar literales, variables y/o funciones, de los tipos reconocidos en la gramática.
8. Debe permitir operadores y operandos, respetando precedencia (usual matemática) y permitiendo el uso de paréntesis.
9. Permitir expresiones aritméticas binarias de suma (+), resta (-), división (/) –entera o decimal según el tipo--, multiplicación (\*), módulo (%) y potencia (^). Para enteros o reales.
10. Permitir expresiones aritméticas unarias de negativo (-); ++ y -- , después del operando; esto para enteros, el negativo adicionalmente se puede aplicar a flotantes. El negativo a literales (puede considerarse parte del número) y el ++ y -- a variables.
11. Permitir expresiones relacionales (sobre enteros y flotantes) de menor, menor o igual, mayor, mayor o igual, igual y diferente. Los operadores igual y diferente permiten adicionalmente tipo booleano.
12. Permitir expresiones lógicas de conjunción (&), disyunción (|) y negación (ésta puede ser tipo función o carácter (! o not)).
13. Debe permitir sentencias de código para las diferentes expresiones mencionadas anteriormente y su combinación, el delimitador de final de expresión será el carácter punto y coma (;). Además, dichas expresiones pueden usarse en las condicionales y bloques de las siguientes estructuras de control.
14. Debe permitir el uso de tipos y la combinación de expresiones aritméticas (binarias y unarias), relacionales y lógicas, según las reglas gramaticales, aritméticas, relacionales y lógicas del Paradigma Imperativo, por ejemplo, tomando como referencia el lenguaje C.
15. La gramática genera un lenguaje con tipado explícito y fuerte.
16. Debe permitir las estructuras de control if-[else], switch, while y for in range (inicio opcional -default 0-, fin requerido, “step” opcional -default 1-), además, permitir return y break. Las expresiones de las condiciones (if,while) deberán ser valores booleanos combinando expresiones aritméticas, lógicas y relacionales.
17. Debe permitir las funciones de leer (enteros y flotantes) y escribir en la salida estándar (cadena carácter, enteros y flotantes), se pueden escribir literales o variables.
18. Debe permitir la creación y utilización de funciones, estos deben retornar valores (entero, char, flotantes y booleanos) y recibir parámetros (tipos excepto string).
    1. La declaración de funciones será de la forma:
       1. func : tipo : id \_ cuerpo \_
       2. Los parámetros se declaran dentro del cuerpo de la función por medio de la palabra reservada param, y no permitirán asignación en la declaración.
       3. En la declaración no se utilizan paréntesis para los parámetros, pero en la invocación si se utilizan, separándolos con comas (,).
19. Debe definir un único procedimiento inicial main, por medio del cual se inicia la ejecución de los programas; con cero o muchas funciones. La declaración del main y las funciones se hace de forma independiente.
20. Además, debe permitir comentarios de una línea (//) o múltiples líneas (/\* \*/).

**Diseño del programa**

La gramática se desarrolló usando la notación Backus Naur estudiada en el curso.

Se siguió el siguiente estándar:

nombreElemento -> expresión/símbolos/otros para los no terminales, terminales, símbolo inicial y producciones, además se decidió nombrar todos los objetos en inglés para así practicar el uso de este y evitar el uso de caracteres especiales propios del español ya que puede incluir problemas de notación que conviene evitar en el desarrollo de una gramática para mantenerla lo más sencilla posible.

**Lista de Terminales**

plus -> '+'

minus -> '-'

product -> '\*'

division -> '/'

power -> '^'

modulus -> '%'

equal -> '='

areEquals -> '=='

gratherThan -> '>'

lessThan -> '<'

gratherEQThan -> '>='

lessEQThan -> '<='

diff -> '!='

unaryPlus -> '++'

unaryMinus -> '--'

main -> 'main'

intT -> 'int'

floatT -> 'float'

stringT -> 'string'

charT -> 'char'

digit -> [0-9]

digitNoZero -> [1-9]

decimal -> \.[digit]+

letter -> [A-Za-z\_]

letterV2 -> [A-Za-z]

true -> 'True'

false -> 'False'

boolT -> 'boolean'

delimeterFunc -> '\_'

endExpr -> ';'

return -> 'return'

lineComment -> '\//'

blockStartComment -> '\/\*'

blockEndCooment -> '\*/'

lineJump -> '\n'

squareParentS -> '['

squareParentE -> ']'

parentS -> '('

parentE -> ')'

coma -> ','

sep -> ':'

loc -> 'local'

global -> 'global'

if -> 'if'

else -> 'else'

switch -> 'switch'

case -> 'case:'

default -> 'default:'

brake -> 'brake'

for -> 'for'

in -> 'in'

range -> 'range'

while -> 'while'

and -> '&'

or -> '|'

print -> 'print'

read -> 'read'

**Lista de no terminales**

id -> letterV2(letter|digit)\*

stringC -> letter(letter|digit)\*

comment -> (lineComment stringC)lineJump| blockStartComment stringC blockEndCooment

types -> charT | stringT | intT | floatT | boolT

typesLim -> charT | intT

typesLimFunc -> charT | intT | floatT | boolT

mathOp -> plus | minus | product | division | modulus | power

boolsMathOp -> lessThan | gratherThan | areEquals | gratherEQThan | lessEQThan | diff

intNum -> minus? digit|digitNoZero digit+

scopeVar -> (loc|global)

floatNum -> minus? intNum decimal digit+

boolNum -> (true|false)

typesNum -> boolNum|intNum|floatNum|stringC|letter

operation -> ((typesNum) mathOp (typesNum) )| (typesNum (unaryMinus | unaryPlus))

extraOperation -> mathOp (typesNum)

extraOperationV2 -> (typesNum) mathOp

boolOp -> (intNum|floatNum) boolsMathOp (intNum|floatNum)

extraBoolOp -> boolsMathOp (intNum|floatNum)

extraBoolOpV2 -> (intNum|floatNum) boolsMathOp

boolOpIds -> id extraBoolOp | extraBoolOpV2 id | id boolsMathOp id

parameter -> typesLim id

parameters -> parentS parameterList? parentE

parameterList -> parameter (',' parameter)\*

**Símbolo Inicial**

si -> expression

**Producciones**

#All about vars

assignation -> equal (boolNum|intNum|floatNum|stringC|operation|letter|arrayElement|boolOp)

varAssig -> scopeVar (varIns|varInsAssig) assignation

varInsAssig -> scopeVar sep types id assignation

varIns -> scopeVar sep types sep id

allTypesVars -> varIns | varInsAssig | varInsAssig | assignation

#All about arrays

assignationArray -> equal squareParentS (intNum|letter (coma intNum|letter)\*)? squareParentE

arrayIns -> scopeVar sep typesLim sep id sep squareParentS intNum squareParentE

arrayInsAssig -> scopeVar sep typesLim sep id sep squareParentS sep intNum sep squareParentE assignationArray

arrayAssig -> (arrayIns|arrayInsAssig) squareParentS intNum squareParentE equal (intNum|letter)

arrayElement -> (arrayIns|arrayInsAssig) squareParentS intNum squareParentE

allTypersArr -> arrayAssig | arrayIns | arrayInsAssig | assignationArray

#I think not necessary

extraVarAssig -> (varAssig|varInsAssig) equal (varAssig|varInsAssig) sep (mathOp (varAssig|varInsAssig)|extraOperation) sep endExpr

boolExtraVarAssig -> (varAssig|varInsAssig) equal (varAssig|varInsAssig) sep (boolsMathOp (varAssig|varInsAssig)|extraBoolOp) sep endExpr

equalizingVars-> (varAssig|varInsAssig) equal (varAssig|varInsAssig|varIns) endExpr

#All the posibilities that a code block can do

allABlockCouldCon -> (varAssig|varIns|varInsAssig|arrayAssig|arrayInsAssig|arrayIns|ifStruc|ifElseStruc|switchStruc|whileLoop|forLopp) endExpr

#All about functions

mainFun -> typesLimFunc sep main sep delimeterFunc sep allABlockCouldCon\* sep return sep (stringC|operation)? sep endExpr sep delimeterFunc

function -> typesLimFunc sep id sep parameters sep delimeterFunc sep allABlockCouldCon\* sep return sep (stringC|operation)? sep endExpr sep delimeterFunc

#conditional structures

ifStruc -> if sep parentS (boolOpIds | boolOp ((or | and) boolOp)\*) parentE delimeterFunc allABlockCouldCon\* (return (stringC|operation)?)? delimeterFunc

ifElseStruc -> ifStruc else delimeterFunc allABlockCouldCon\* (return (stringC|operation)?)? delimeterFunc

switchStruc -> switch parentS (intNum|stringC) parentE delimeterFunc (case(intNum|letter) allABlockCouldCon\* brake endExpr)+ sep default allABlockCouldCon\* brake endExpr delimeterFunc

#loops

whileLoop -> while parentS (boolOpIds | boolOp ((or | and) boolOp)\*) parentE delimeterFunc allABlockCouldCon\* return? (stringC|operation)? delimeterFunc

forLopp -> for id sep in sep range parentS (0|intNum) coma intNum (1|intNum) parentE delimeterFunc allABlockCouldCon\* return? (stringC|operation)? delimeterFunc

#StandardIO

printLn -> print parentS (stringC | operation | boolNum | intNum | floatNum | id) parentE endExpr

readInput -> read parentS (stringC | intNum | floatNum) parentE endExpr

#Experiment

id -> allTypersArr | allTypesVars | function | main

#All the connections

expression -> main

expression -> function

expression -> id

expression -> and

expression -> or

expression -> typesNum

expression -> printLn

expression -> readInput

expression -> comment

expression -> expression extraOperation

expression -> extraOperationV2 expression

expression -> expression extraBoolOp

expression -> extraBoolOpV2 expression

expression -> id mathOp id

expression -> id boolsMathOp id

expression -> id mathOp expression

expression -> id boolsMathOp expression

expression -> operation

expression -> boolOp

expression -> extraOperation

expression -> extraOperationV2

expression -> extraBoolOp

expression -> extraBoolOpV2

expression -> ifStruc | ifElseStruc | switchStruc

expression -> whileLoop | forLopp

expression -> expression mathOp

expression -> expression boolNum

expression -> expression boolsMathOp

expression -> expression mathOp expression

expression -> expression boolNum expression

expression -> expression boolsMathOp expression

expression -> expression expression mathOp expression

expression -> expression expression boolNum expression

expression -> expression expression boolsMathOp expression

expression -> expression expression

expression -> -expression

expression -> (expression)

La lista completa de terminales, no terminales y producciones se encuentra en el documento adjunto llamado “gramatica BNF.txt” que se encuentra en el repositorio del proyecto, pueden ser consultadas con mejor detalle en el mismo y además se encuentran agrupadas por su tipo y uso.

**Análisis de resultados**

| **Objetivo** | **Estado** | **Lección** |
| --- | --- | --- |
| Creación de funciones | 100% |  |
| Manejar tipos | 100% |  |
| Manejar Variables | 100% |  |
| Manejar expresiones | 100% |  |
| Creación de variables | 100% |  |
| Crear arreglos | 100% |  |
| Definir alcance de variables | 100% |  |
| Separador 100% | 100% |  |
| Combinación de expresiones | 100% |  |
| Operadores y operandos | 100% |  |
| Operaciones aritméticas | 100% |  |
| Operaciones aritméticas Unarias | 100% |  |
| Expresiones relacionales | 100% |  |
| Expresiones lógicas | 100% |  |
| Tipado | 100% |  |
| Estructuras de control | 100% |  |
| Entrada y salida estándar | 100% |  |
| Procedimiento inicial main | 100% |  |
| Comentarios | 100% |  |

**Bitácora**

La bitácora fue generada de manera automática usando el sistema de control de versiones Git, puede consultar el repositorio online de la herramienta accediendo al siguiente enlace:

https://github.com/FR3DD221/Tarea-1\_Compiladores.git