CURSO 2022-2023

METALENGUAJES

DISEÑO DE LENGUAJES DE PROGRAMACIÓN

SARA FERNÁNDEZ ARIAS
UO269546

CONVOCATORIA EXTRAORDINARIA DE JUNIO

Contenido

Léxico del lenguaje

Descrito mediante expresiones regulares.

INT_CONSTANT	**[0-9]+**
REAL_CONSTANT	[0-9]+'.'[0-9]+
CHAR_CONSTANT	'\'\\n\'' ‘\''~[\t\r\n]'\''
IDENT	[a-zA-Z]+(('_')?[a-zA-Z0-9])*
LINE_COMMENT	'//' .*? ('\n' | EOF) -> skip
MULTILINE_COMMENT	‘/*' .*? '*/' -> skip
WHITESPACE	[\t\r\n]+ -> skip

Sintaxis del lenguaje

Descrita mediante una Gramática Libre de Contexto.

start-> definition* **EOF**

definition-> **IDENT ‘(‘**parameter*’**)’ ‘:’** type **‘{‘**varDefinition* statement*’**}’**

| **‘var’ IDENT ‘:’** type ‘**;**’

|**’struct’ IDENT ‘{‘(**structField ‘;’) * **‘}’ ‘;’**

parameter-> **IDENT ‘:’** type

structField-> **IDENT ‘:’** type

type-> **‘int’**

|**’float’**

|**’char’**

| **‘[‘INT_CONSTANT ‘]’** type

| **IDENT**

statement -> expression **'='**expression **';'**
| **'if'**'**(**'expression'**)**''{‘statement* '}’(‘**else**' '**{**' statement*'**}**')?

| **'while'** '**(**‘expression'**)**' '**{**' statement* '**}**'

| '**read**' expression '**;**'
| ('**print**' **|**'**printsp**'**|**'**println**’) expression? '**;**'

| **IDENT** '**(**'(expression (','expression) *)?')' '**;**'

| '**return**' expression **‘;’**
expression-> **IDENT**

| **INT_CONSTANT**

|**REAL_CONSTANT**

|**CHAR_CONSTANT**

|’ **(**‘expression’**)’**

|expression ‘.’ **IDENT**

| expression ‘**[‘**expression’**]’**

|**’!’** expression

| **‘<**’type **‘>’’(**‘expression**’)’**

|**IDENT** ‘(‘(expression (‘,’ expression) *)? ‘)’

|expression ('*'|'/'|'%') expression

|expression ('+'|'-’) expression

|expression ('>'|'<'|'>='|'<='|'=='|'!=') expression

| expression '&&'expression

| expression '||'expression

Gramática Abstracta

Detalle de los nodos del AST.

Program -> definitions:definition*;

VarDefinition:definition -> type name:string;

FunctionDefinition:definition -> *name:*string *parameters:*varDefinition* *returnType*:type *localDefs*:varDefinition* *body*:statement*;

StructDefinition:definition ->*name*:string *fields*:StructField*;

StructField->*name*:string type;

VoidType:type->;

IntType:type -> ;

FloatType:type -> ;

CharType:type -> ;

ArrayType:type -> *dimension:*int type;

StructType:type -> *name:*string;

Print:statement -> expression *variant:*string;

Read:statement -> expression;

IfStatement:statement -> condition:expression *body*:statement* *elseBody*:statement*;

While:statement ->*condition:*expression *body:*statement*;

Assignment:statement -> *left:*expression *right*:expression;

Return:statement -> expression;

InvocationStatement:statement -> *name*:string *parameters:*expression*;

Invocation:expression -> *name*:string *parameters:*expression*;

ArithmeticExpression:expression -> *left*:expression *operator*:string *right:*expression;

Comparison:expression -> *left*:expression *operator:*string *right:*expression;

And:expression -> *left*:expression *right:*expression;

Or:expression -> *left*:expression *right:*expression;

Not:expression -> expression;

Cast:expression -> type expression;

ArrayAccess:expression -> *array:*expression *position:*expression;

StructFieldAccess:expression -> *struct:*expression *field:*string;

VariableReference:expression -> *name:*string;

LiteralInt:expression -> *value:*int;

LiteralFloat:expression -> *value:*double;

LiteralChar:expression -> value:character;

Fase de Comprobación de tipos.

Descrita mediante una gramática atribuida.

Nodo	Predicados	Reglas Semánticas
Program → _definitions_:definition*		
variable → _name_:String _type_:type		
StructField → _name_:String _type_:type		
VarDefinition:definition → _type_:type _name_:String		
FunctionDefinition:definition → _name_:String _parameters_:varDefinition* _returnType_:type _localDefs_:VarDefinition* _statements_:statement*	hasSimpleType(parameters)==true hasSimpleType(returnType)==true	
StructDefinition:definition → _name_:String _fields_:StructField*		
VoidType:type → λ		
IntType:type → λ		
FloatType:type → λ		
CharType:type → λ		
ArrayType:type → _dimension_:String _type_:type		
StructType:type → _name_:String		
Print:statement → _expression_:expression _variant_:String	hasSimpleType(expression.type)== true	print.hasReturnStatement=false
Read:statement → _expression_:expression	hasSimpleType(expression.type)==true expression.lvalue==true	read.hasReturnStatement=false

IfStatement:statement → _condition_:expression _body_:statement* _elseBody_:statement*	Condition.type==intType	If(elseBody.size>0){
		ifStatement.hasReturnStatement=
		((Body.stream().AnyMatch(stmt->stmt.hasReturnStatement))
		&&(ElseBody.stream().AnyMatch(stmt->stmt.hasReturnStatement))
		}else{
		ifStatment.hasReturnStatement=false
		}

While:statement → _condition_:expression _body_:statement*	Condition.type==IntType	While.hasReturnStatement= Body.stream().AnyMatch(stmt->stmt.hasReturnStatement)
Assignment:statement → _left_:expression _right_:expression	hasSimpleType(left.type) left.type,==right.type left.lValue==true	Assignment.hasReturnStatement=false
Invocation:statement → _name_:String _parameters_:variable*	checkArguments(functionDefinition.parameters, parameters)	Invocation.hasReturnStatement=false

Invocation:expression → _name_:String _parameters_:variable*	checkArguments(functionDefinition.parameters, parameters)
	functionDefinition.returnType!=VoidType
		Invocation.type=FunctionDefinition.returnType
		Invocation.lvalue=false

Return:statement → _expression_:expression	If(Return.functionDefinition.returnType!=void){	Return.functionDefinition=param
		return.hasReturnStatement=true

	Return.functionDefinition.returnType
	==expression.type
	}else{
	Expression==null
	}	

ArithmeticExpression:expression → *left*:expression *operator*:String *right*:expression	left.type==right.type
	left.type!=VoidType
		ArithmeticExpression.type=left.type
		ArithmeticExpression.Lvalue=false

Comparison:expression → *left*:expression *operator*:String *right*:expression	left.type=right.type
	left.type=IntType ||
	left.type=FloatType
		Comparisson.type=left.type
		Comparison.Lvalue=false

And:expression → *left*:expression *right*:expression	left.type=right.type
	left.type==IntType
		And.type=left.type
		And.Lvalue=false

Or:expression → *left*:expression *right*:expression	left.type=right.type
	left.type==IntType
		Or.type=left.type
		Or.Lvalue=false

Not:expression → *expression*:expression	expression.type==IntType	Not.type=expression.type
		Not.Lvalue=false

Cast:expression → *type*:type *expression*:expression	type!=expression.type
	hasSimpleType(type)
	hasSimpleType(expression.type)
		cast.type= type
		Cast.Lvalue=false

ArrayAccess:expression → *array*:expression *position*:expression	array.type==ArrayType
	position.type==IntType
		ArrayAccess.type= array.type
		ArrayAccess.Lvalue=true

StructFieldAccess:expression → *struct*:expression *field*:String	Struct.type==StructType
	foundField(stuct.type.getFields(),field) ==true
		structFieldAcces.type=struct.definition.fields.stream().find(field->field.name.equals(field)).type
		StructFieldAccess.Lvalue=true

VariableReference:expression → *name*:String		variableReference.type=expression.definition.type
		VariableReference.lValue=true

LiteralInt:expression → *value*:String		literalInt.type=IntType
		LiteralInt.Lvalue=false

LiteralFloat:expression → *value*:String		literalFloat.type=FloatType
		LiteralFloat.Lvalue=false

LiteralChar:expression → *value*:String		literalChar.type=CharType
		LiterealChar.Lvalue=false

Atributos

Nodo/Categoría Sintáctica	Nombre del Atributo	Tipo Java	Heredado /Sintetizado	Descripción
Variable	Type	Type	Heredado	Las variables reciben el tipo de su definición
Expression	Type	Type	Sintetizado	Las expresiones tendrán un atributo tipo que indica qué operaciones admiten. Este dependerá a su vez de los tipos de las expresiones que las conforman.
ReturnStatement	FunctionDefinition	Type	heredado	El nodo funciónDefinition **se enviará a si mismo haciendo uso del parámetro del visitor**) a su hijo, el nodo returnStatement.
Expression	lValue	Boolean	Sintetizado	Describe si puede modificarse su valor, si puede aparecer a la izquierda en una asignación.

Statement	HasReturnStatement	Boolean	Sintetizado	Utilizo este atributo para comprobar que las definiciones de funciones cuyo tipo de retorno no es void contienen un return statement. **Este atributo ha sido introducido para facilitar el trabajo en codeGeneration.** Este permite comprobar facilmente que una función con voidType no tiene sentencia de retorno explícita (sin expresión) **.**

Métodos auxiliares

```
Private Boolean hasSimpleType(expression e){

if(e.type==ArrayType || e.type==StructType)

        return false

return true

}

 private boolean checkArguments(List<Variable> parameters, List<Expression> parametersGot) {

            Variable currentExpected;

            Expression valueRecieved;

            for (int i = 0; i < parameters.size(); i++) {

              currentExpected=parameters.get(i);

              valueRecieved=parametersGot.get(i);

              if(!currentExpected.getType().equals(valueRecieved.getType())){

                return false;  } }

            return true;   }

 private StructField foundField(List<StructField> fields,String fieldToFind){

            for (StructField field:fields   ) {   if(field.getName().equals(fieldToFind)){   return field;   }    }

        return null;

         }
```

Fase de generación de Código

Descrita mediante una especificación de código

Función	Plantillas de Código
run[[Program]]	**run**[[Program → definitions:definition*]] = **#SOURCE {sourceFile}** CALL main HALT **define**[[definitions]]

define[[definition]]	**define**[[VarDefinition → type:type name:String]] =
	if(VarDefinition.isGlobal)
	  #GLOBAL {name}: {type}
	Else
	  ‘{name}:{type}
	define[[StructDefinition → name:String fields:StructField*]] =
	‘Struct {name}
	define[[fields]]
	define[[StructField → name:String type:type **]]=**
	‘{name}:{type}
	define[[FunctionDefinition → name:String parameters:varDefinition*
	  returnType:type localDefs:VarDefinition* statements:statement*]] =
	#LINE {start.line}
	{name}:
	‘*Parameters*
	  define[[parameters]]
	‘*Local Variables:*
	  define[[localDefs]]
	  ENTER {-localDefs.get(localDefs.size -1).direction }
	‘*Body*
	  #LINE {statements.start.line}
	  execute[[statements]]

```
                    if(!hasReturnStatement && returnType==VoidType)
                            if(parametersSize+returnType.size+-localDefs.get(localDefs.size -
                            1).direction ==0)

                              RET

                      Else

                           RET {parametersSize},{returnType.size},{-localDefs.get(localDefs.size
                            -1).direction }
address[[variable]]   address[[variableReference → name:String type:type ]] =

                    if(! global) :
                      PUSH BP
                      PUSHA {variable.definition.direction}
                     ADDI
                    Else:
                      PUSHA {variable.definition.direction}
                    address[[arrayAccess → array:expression position:expression ]] =


                    address[[array]]
                    value [[position]]
                    PUSHI {arrayAccess.type.size}
                    MULI
                    ADDI
                    address[[structFieldAccess]]=

                    address[[struct]]
                    PUSHI {struct.definition.getField(field).direction}
                    ADDI

execute[[statement]]  execute[[Print → expression:expression variant:String ]] =

                      value[[expression]]

                      If(variant==’sp’)

                          OUT{expression.type.suffix}+ 32

                      If(variant==’ln’)

                         OUT{expression.type.suffix}+10

                      else

                        OUT{expression.type.suffix}

                    execute[[Read → expression:expression ]] =
  Address[[expression]]
  IN{expression.type.suffix}
 STORE{expression.type.suffix}

execute[[IfStatement → condition:expression body:statement* elseBody:statement*
 ]] =
  value[[condition]]
  jz elseBody{label}
  execute[[body]]
  jmp endIf{label}
  elseBody{label}
   #LINE {elseBody.start.line}
   execute[[elseBody]]
  endIf{label}

execute[{While → condition:expression body:statement*]]

  whileStart{label}
  value[[condition]]
  jz whileEnd{label}
   #LINE {body.start.line}
  execute[[body]]
  jmp whileStart{label}
  whileEnd{label}

execute[[Assignment → left:expression right:expression ]] =
   address[[left]]
   value[[right]]
  store{left.type.suffix}
```

```
execute[[Return → expression:expression ]] =

  value[[expression]]

      if(functionDefinition.parametersSize+ functionDefinition returnType.size+-
           functionDefinition localDefs.get(localDefs.size -1).direction ==0)

           RET

    Else

        RET {functionDefinition. parametersSize},

                { functionDefinition.returnType.size},

                {-functionDefinition.localDefs.get(localDefs.size -1).direction }


execute[[InvocationStatement → name:String parameters:expression
*]] =

   value[[parameters]]

   CALL {name}

   If(definition.getReturnType()!=VoidType)

   POP{ definition.returnType.suffix}
value[[expression]]   value[[Invocation → name:String parameters:expression* ]] =

  value[[parameters]]

  CALL {name}
value[[ArithmeticExpression → left:expression

                              operator:String right:expression ]] =

  value[[left]]

  value[[right]]

  if(operator=='+')

      ADD{left.type.suffix}

  if(operator=='-')

       SUB{left.type.suffix}
  if(operator==’*’)
        MUL{left.type.suffix}
  if(operator==’/’)
        DIV{left.type.suffix}
value[[Comparison → left:expression
                                              operator:String  right:expression ]] =
    value[[left]]
    value[[right]]
  if(operator==’>’)
       GT{left.type.suffix}
   if(operator==’>=’)
        GTE {left.type.suffix}
    if(operator==’<’)
        LT {left.type.suffix}
    if(operator==’<=’)
        LTE {left.type.suffix}
value[[And → left:expression  right:expression ]] =
    value[[left]]
    value[[right]]
    AND
value[[Or → left:expression  right:expression ]] =
    value[[left]]
    value[[right]]
    OR
value[[Not → expression:expression ]] =
    value[[expression]]

    NOT
value[[Cast → type:type  expression:expression ]] =
    value[[expression]]

    If(type == float  && expression.type== char ||

            type=char && expression.type== float){

                    {expression.type.suffix}2i

                     i2{type.suffix}

   }   else{

    {expression.type.suffix}2{type.suffix}

  }
value[[ArrayAccess → array:expression      position:expression]]=

       address[[array]]

       value[[position]]

       LOAD {array.type.suffix}

value[[StructFieldAccess → struct:expression   field:string]]

      address[[structFieldAccess]]

      LOAD{struct.definition.type.suffix}

value[[VariableReference → name:String ]] =

      address[[variableReference]]

      LOAD{variableReference..type.suffix}
value[[LiteralInt → value:String ]] =

     PUSHI {value}
value[[LiteralFloat → value:String ]] =

     PUSHF {value}

value[[LiteralChar → value:String ]] =

     PUSHB {value}
```