Práctica 4

Asignatura PL
Curso GII (2019-2020)
Profesor José Luis Sierra Rodríguez
Alumnos Alejandro Cancelo Correia
Tomás Golomb Durán

https://github.com/AlexCCo/Simpler-Compiler-PL

INTRODUCCIÓN

En esta fase de desarrollo lo que haremos será implementar una sintaxis abstracta para nuestro lenguaje.

Para ello especificaremos una gramática de atributos que nos permitirá construir el “constructor de árboles abstractos”.
Así mismo implementaremos una definición de constructoras y un diagrama de clases para finalizar la creación de nuestra sintaxis abstracta.

Como recordatorio introduciremos la definición del lenguaje y la GIC que empleamos para la creación del analizador sintáctico (también conocido como Parser)

Definición del lenguaje

Los **programas** están **formados por**:
(i) una sección de declaraciones
(ii) una sección de instrucciones.

Ambas secciones están **separadas por &&.**

La sección de **declaraciones**, por su parte, está **compuesta** por **una o más declaraciones, separadas por puntos y coma**.
Cada declaración consta de:
(i) un nombre de tipo
(ii) un nombre de variable.

Los nombres de **tipo** pueden ser **int, real y bool** y los nombres de **variable comienzan** necesariamente **por una letra, seguida de una secuencia de cero o más letras, dígitos, o subrayado (_).**

La sección de **instrucciones** consta de **una o más instrucciones separadas por puntos y coma**. El lenguaje únicamente considera **instrucciones de asignación.** Dichas instrucciones constan de una **variable, seguida de =, seguida de una expresión**.

Las **expresiones básicas** consideradas son **números enteros con y sin signo, números reales con y sin signo, variables, y true y false**.

Los **números enteros** comienzan, opcionalmente, con un signo + o - y seguidamente debe aparecer una secuencia de 1 o más dígitos (no se admiten ceros no significativos a la izquierda).

Los **números reales** tienen, obligatoriamente, una parte entera, cuya estructura es como la de los números enteros, seguida de bien una parte decimal, bien una

parte exponencial, o bien una parte decimal seguida de una parte exponencial. La parte decimal comienza con un ., seguido de una secuencia de 1 o más dígitos (no se permite la aparición de ceros no significativos a la derecha). Por último, y también opcionalmente, puede aparecer una parte exponencial (e o E, seguida de un exponente, cuya estructura es igual que la de los números enteros).

Los **operadores** que pueden utilizarse en las expresiones son los operadores **aritméticos binarios** usuales (+,-,* y /), el menos **unario** (-), los **operadores lógicos** (and, or y not) y los **operadores relacionales** (<,>,<=,>=,==,!=).

También es posible utilizar paréntesis para alterar las precedencias y asociatividades de los operadores.

Gramática s-atribuida descendente

La **gramática** está definida de la siguiente forma:

G = {V, T, P, S}

Donde:

V = {S, SD, VAR, INST, IS, E0, R0, opE0, E1, R1, opE1, E2, R2, opE2, E3, R3, opE3, E4, opE4}
Conjunto de *símbolos no terminales*.

T = {NE, NR, NV, EOI, EOD, EOF, TP, OpAssig, true, false, and, or, <, <=, >, >=, ==, !=, *, /, not, -, +}
Conjunto de *símbolos terminales* (lo que representan está al final del documento)

S es el *símbolo inicial*

P:
Conjunto de reglas de producción e ilustrado en las siguientes líneas.

** Las reglas ya están acondicionadas de la práctica anterior*

S ---> SD && IS
S.a = Program(SD.a, IS.a)

SD ---> VAR RSD
RSD.ah = VAR.a
SD.a = RSD.a
RSD ---> ; VAR RSD
RSD1.ah = DecComp(VAR.a, RSD0.ah)
RSD0.a = RSD1.a
RSD ---> ε

https://github.com/AlexCCo/Simpler-Compiler-PL

RSD.a = RSD.ah

VAR -> tp nv
VAR.a = DecSimple(tp.lex, nv.lex)

IS ---> INST RIS
RIS.ah = INST.a
IS.a = RIS.a
RIS ---> ; INST RIS
RIS1.ah = InstComp(INST.a, RIS0.ah)
RIS0.a = RIS1.a
RIS ---> ε
RIS.a = RIS.ah
INST ---> nv = E0
INST.a = InstSimple(nv.lex, E0.a)

// Factor común
E0 ---> E1 R0
R0.ah = E1.a
E0.a = R0.a
R0 ---> + E0
R0.a = exp(R0.ah, +, E0.a)
R0 ---> - E1
R0.a = exp(R0.ah, -, E1.a)
R0 ---> ε
R0.a = R0.ah

//
E1 ---> E2 R1
R1.ah = E2.a
E1.a = R1.a
R1 ---> opE1 E2 R1
R11.ah = exp(R10.ah, opE1.a, E2.a)
R10.a = R11.a
R1 ---> ε
R1.a = R1.ah

//
E2 ---> E3 R2
R2.ah = E3.a
E2.a = R2.a
R2 ---> opE2 E3 R2
R21.ah = exp(R20.ah, opE2.a, E3.a)
R20.a = R21.a
R2 ---> ε

https://github.com/AlexCCo/Simpler-Compiler-PL

R2.a = R2.ah

// Factor común
E3 ---> E4 R3
R3.ah = E4.a
E3.a = R3.a
R3 ---> opE3 E4
R3.a = Exp(R3.ah, opE3.a, E4.a)
R3 ---> ε
R3.a = R3.ah

E4 ---> opE4 E5
E4.a = exp(opE4.a, E5.a)
E4 ---> E5
E4.a = E5.a

E5 ---> terminal
E5.a = terminal.a
E5 ---> (E0)
E5.a = E0.a

terminal ---> NV
terminal.a = NV.lex
terminal ---> NE
terminal.a = NE.lex
terminal ---> NR
terminal.a = NR.lex
terminal ---> true
terminal.a = true
terminal ---> false
terminal.a = false

opE1 ---> and | or
opE2 ---> < | <= | > | >= | == | !=
opE3 ---> * | /
opE4 = not | -

Funciones semanticas

op ---> string
arg1 ---> E
return ---> EMono

```
func exp(op, arg1){
  switch(op){
```

https://github.com/AlexCCo/Simpler-Compiler-PL

```
    case not: return not(arg1);
    case -: return negativo(arg1);
  }
}
```

```
op           ---> string
arg1, arg2 ---> E x E
return       ---> EBin

funct exp(arg1, op, arg2){
  switch(op){
    case +: return suma(arg1, arg2);
    case -: return resta(arg1, arg2);
    case <: return menor_q(arg1, arg2);
    case <=: return menor_iq(arg1, arg2);
    case >: return mayor_q(arg1, arg2);
    case >=: return mayor_iq(arg1, arg2);
    case ==: return comp(arg1, arg2);
    case !=: return distinto(arg1, arg2);
    case and: return and(arg1, arg2);
    case or: return or(arg1, arg2);
  }
}
```

Constructores

Program(a, b): Dec x Inst -> Program

DecComp(a, b): DecComp x Dec -> Dec

DecSimple(a) string x string -> Dec

InstSimple(a) string x E -> InstSimple

InstComp(a, b): InstComp x Inst -> Inst

Suma(a, b): E x E -> E

Resta(a, b): E x E -> E

Menor(a, b): E x E -> E

https://github.com/AlexCCo/Simpler-Compiler-PL

MenorIgual(a, b): E x E -> E

Mator(a, b): E x E -> E

MayorIgual(a, b): E x E -> E

Comparacion(a, b): E x E -> E

Distinto(a, b): E x E -> E

And(a, b): E x E -> E

Or(a, b): E x E -> E

Not(a): E -> E

Negativo(a): E -> E

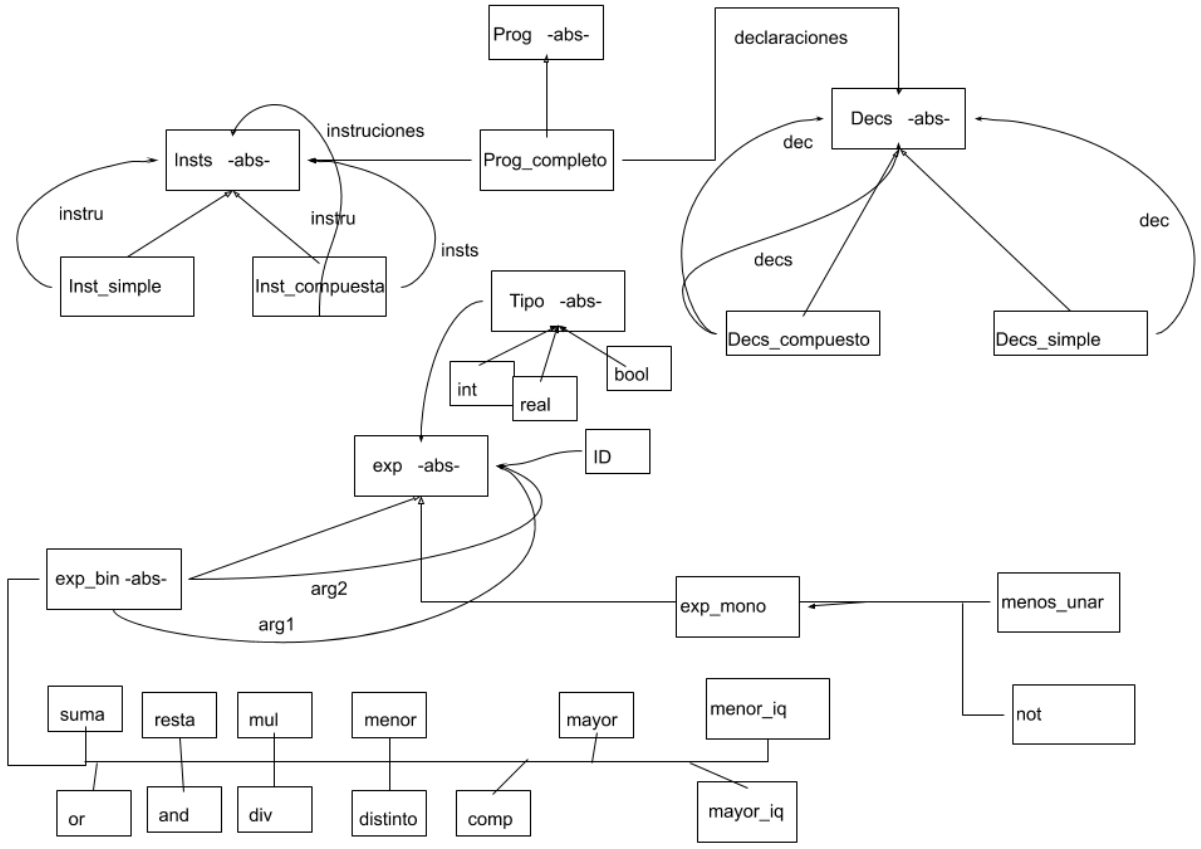
NV(a): String -> E

NR(a): String -> E

NE(a): String -> E

boolean(a): String -> boolean

Diagrama de clase



Gramática de s-atribuidas ascendente

S ---> SD && IS
S.a = Prog_completo(SD.a, IS.a)

SD ---> SD **;** VAR
SD0.a = decs_compuesta(SD1.a, VAR.a)
SD ---> VAR
SD.a = VAR.a
VAR -> TP NV
VAR.a = decl(TP.a, NV.a)

IS ---> IS **;** INST
IS.a = inst_compuesta(IS.a, INST.a)
IS ---> INST

https://github.com/AlexCCo/Simpler-Compiler-PL

IS.a = INST.a
INST ---> NV = E0
IS.a = instuc(NV.a, E0.a)

E0 ---> E1 **+** E0
E00.a = suma(E1.a, E01.a)
E0 ---> E1 **-** E1
E0.a = resta(E10.a, E11.a)
E0 ---> E1
E0.a = E1.a

E1 ---> E1 opE1 E2
E1.a = exp(opE1, E10.a, E2.a)
E1 ---> E2
E1.a = E2.a

E2 ---> E2 opE2 E3
E2.a = exp(opE2, E2.a, E3.a)
E2 ---> E3
E2.a = E3.a

E3 ---> E4 opE3 E4
E3.a = exp(opE3, E40.a, E41.a)
E3 ---> E4
E3.a = E4.a

E4 ---> opE4 E5
E4.a = exp(opE4, E5.a)
E4 ---> E5
E4.a = E5.a
E5 ---> terminal
E5.a = terminal.a
E5 ---> (E0)
E5.a = E0.a

terminal ---> NV
terminal.a = NV(NV.lex)
terminal ---> NE
terminal.a = NE(NE.lex)
terminal ---> NR
terminal.a = NR(NR.lex)
terminal ---> true
terminal.a = boolean(true.lex)
terminal ---> false
terminal.a = boolean(false.lex)