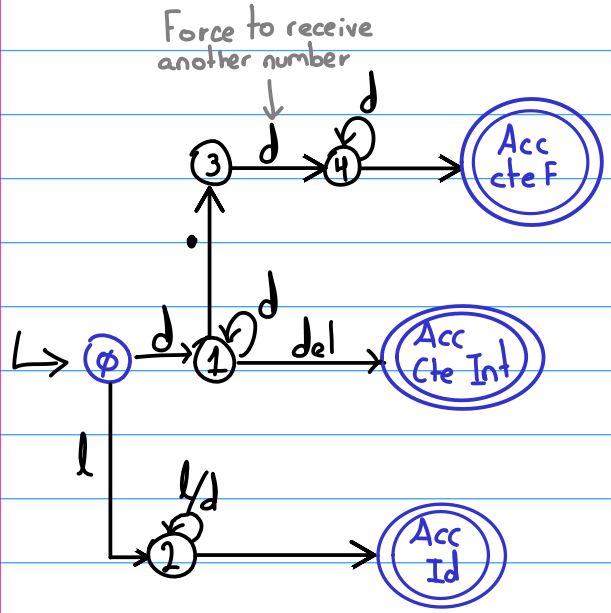
Compiladores

2025/09/29

white space

Let $d = \{0, \dots, 9\}$ let $l = \{a, \dots, z\}$ $del = \{\not b, ret, +, *, ;, \dots, \}$



2025/09/29

Defining the sets helps reducing the complexity of the matrix

1235 → dec ✓
0101B → bin ✓
0101 → dec ✓
35BEh → hex ✓
ØABH → hex ✓
ABH → ID
float ✓

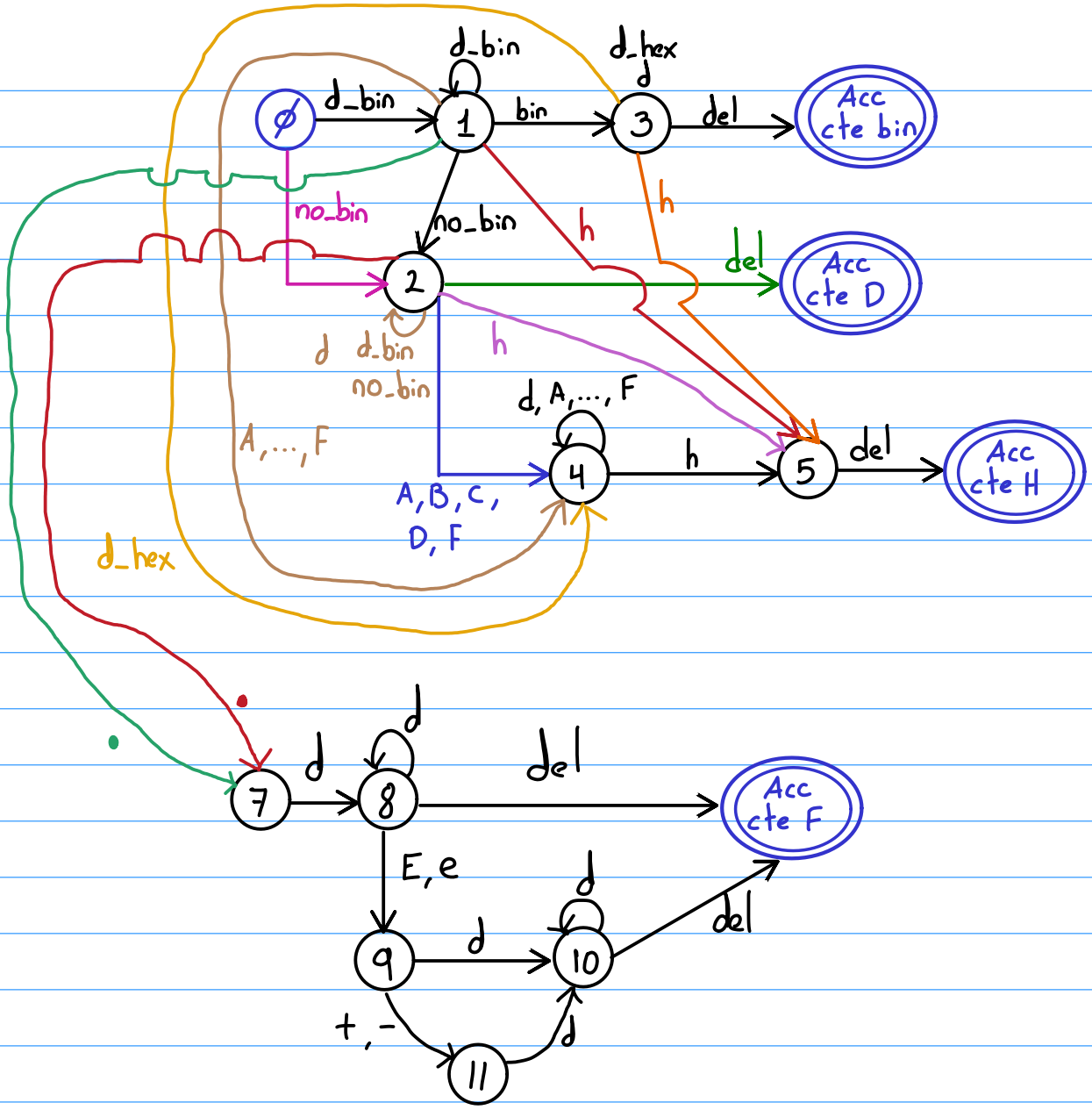
sets $d = \{0, \dots, 9\}$
$d\_bin = \{0, 1\}$
$no\_bin = \{2, \dots, 9\}$
$l = \{a, \dots z, A, \dots, Z\}$
$bin = \{B, b\}$
$Hex = \{H, h\}$
$d\_hex = \{A, C, D, E, F\} + bin + d - Bb - Ee$

In hexadecimal B is special because it is the binary delimiter and E is special as well because it is used in scientific notation.

- Constants in scientific notation do have points
- This DFA doesn't have memory, it doesn't know the previous states
- The more freedom and flexibility in sintax you give to the user, the more complex is the final states diagram

Finish all the functions with the transition map

Add float numbers and scientific notation


d_bin
d_hex
d
d_bin
bin
del
Acc
cte bin
no_bin
no_bin
h
h
del
Acc
cte D
d
d_bin
no_bin
h
d, A,..., F
A,...,F
A,B,C,
D, F
h
del
Acc
cte H
d_hex
d
d
del
Acc
cte F
E, e
d
d
del
+, -
d

Sintax diagrams

Notation

tokens / terminal symbols

sintactic variables / non-terminal symbols

this is a way to summarize modules

flux (direction), this is unidirectional

<rule>

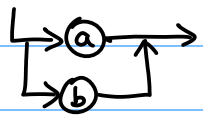
rule development

Example

<X>

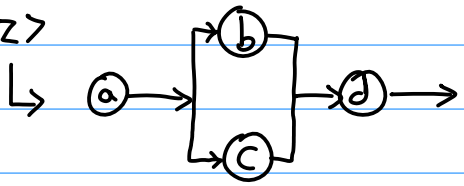
$L = \{a\}$

<Y>



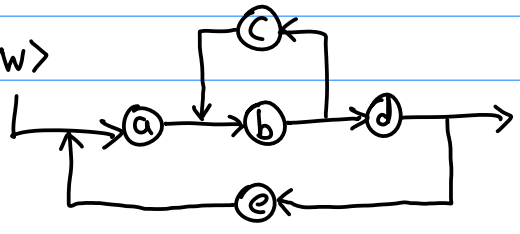
$L = \{a, b\}$

<Z>



$L = \{abd, acd\}$

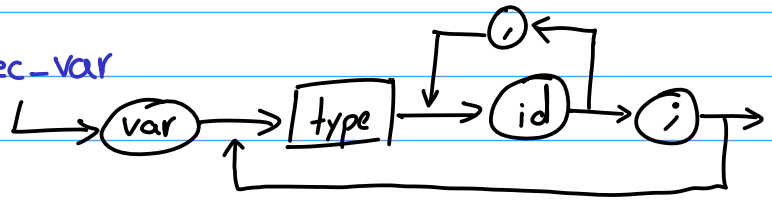
<W>



... incomplete ...

2 different styles to declare variables

dec_var



dec_var (2)

Formal Grammars

$\alpha \rightarrow \beta$ alpha produces beta

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

$A = \cdots \quad C * D$

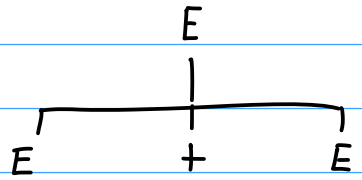
Can we generate id + id * id given the set of rules of E?

$E \Rightarrow E + E$
$\Rightarrow id + E$
$\Rightarrow id + E * E$
$\Rightarrow id + id * E$
$\Rightarrow id + id * id$ ✓

$E \Rightarrow E * E$
$\Rightarrow E + E * E$
$\Rightarrow id + E * E$
$\cdots$
$\Rightarrow id + id * id$ ✓

The root is the sentential grammar symbol

- Sequence is given by the leafs read from left to right

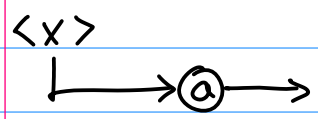


ε : epsilon that represents an empty set (∅)

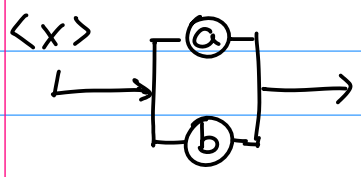
Z : capital z

z : lowercase z

$: end of file

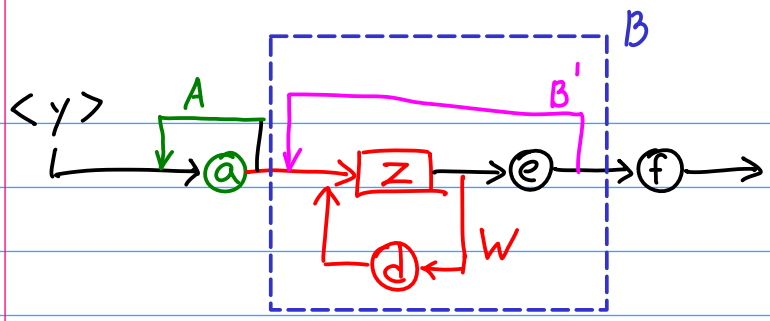


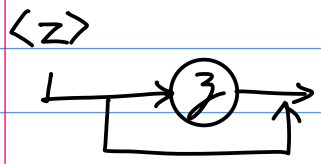
X → a read as "X produces a"



X → a

X → b





$Z \to z$

$Z \to \varepsilon$

Z produces z (lowercase Z) or nothing

$A \to a$

$A \to aA$ $\equiv$ $A \to aA'$, $A' \to \varepsilon$, $A' \to <A>$

$B \to WeB'$

$B' \to \varepsilon$

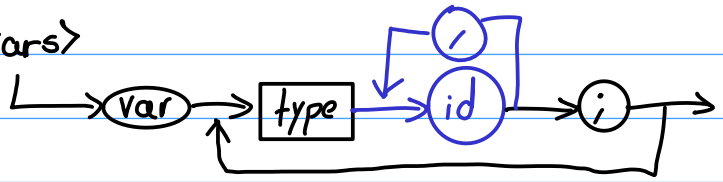
$B' \to B$

$W \to <z, w'>$

$w' \to$

Both expressions are equivalent, but the right option is better

<vars>



<vars> → var <DEC_VAR>

<DEC_VAR> → <TYPE> <LIST_id> ; <DV'>

could be 1 or more ids

cloud cycle or not

<DV'> → ε

<DV'> → <DEC_VAR>

2025/10/09

Brute-Force method

It supposes that every sintactic variable...

Another module involves a lexic

This is a very simple method (our brain uses this method) but at the same time is very dangerous because the probability of getting errors is very high.

It is a lineal method because it uses a brute-force approach

Example

This is just 1 sintactic variable, we define a function per every sintactic variable. The description of B is missing.

A → aB

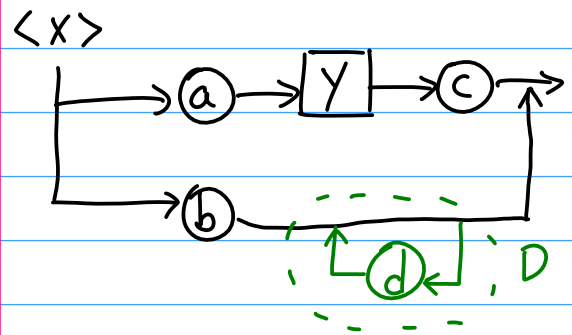
A → c

↓

```
void A()
{ if
```

Example

1st step: get grammar rules

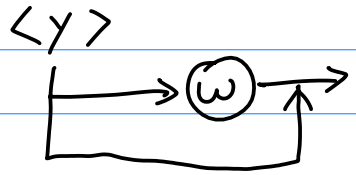


$X \rightarrow aYc$

$X \rightarrow bD$

$D \rightarrow \varepsilon$

$D \rightarrow dD$



$Y \rightarrow \omega$

$Y \rightarrow \varepsilon$

Programming function Y()

```
void Y()
{ if (next_token == 'ω')  // we consume the token
    {go_ahead(); // token was consumed
    }
}
```

Programming function D()

```
void D()
{ while (next_token == 'd')  // we consume the token if it is 'd'
   { go_ahead();
   }
}
```

Programming function X()

```
void X()
{ if (next_token == 'a')
  { go_ahead();
    Y();
    if (next_token != 'c')  // Y() must return the token 'c', otherwise it is an error
    { Error('c');
    }
    else
       { go_ahead(); }  // received 'c' as the next_token, continue
  }
  else
  { if (next_token == 'b')
    { go_ahead();
      D();
    }
  else
     Error(a,b)  // you have either a or b, not anything else
```

$X \to aYb$

$X \to dw$

$X \to aB$

we have 2 alternatives of the same variable that start from the same place, this is not supported, apply operations over grammar called "common left factor", factorize it

$X \to dw$

$X \to aX'$ X' is the uncommon part

$X' \to Yb$

$X' \to B$

Common Left Factor

$\alpha \to \beta\theta$

$\alpha \to \beta\sigma$ $\Rightarrow$

$\alpha \to \omega$

$\alpha \to \omega$

$\alpha \to \beta\alpha'$

$\alpha' \to \theta$

$\alpha' \to \sigma$

Programming function X()

```
void X()
{if (next_token =='d')
   { go_ahead();
     if (next_token == 'w')
       { go_ahead(); }
     else
       {Error}
   }
  if (next_token != 'a')
    {Error}
  else
    { go_ahead();
      x'();
    }
```

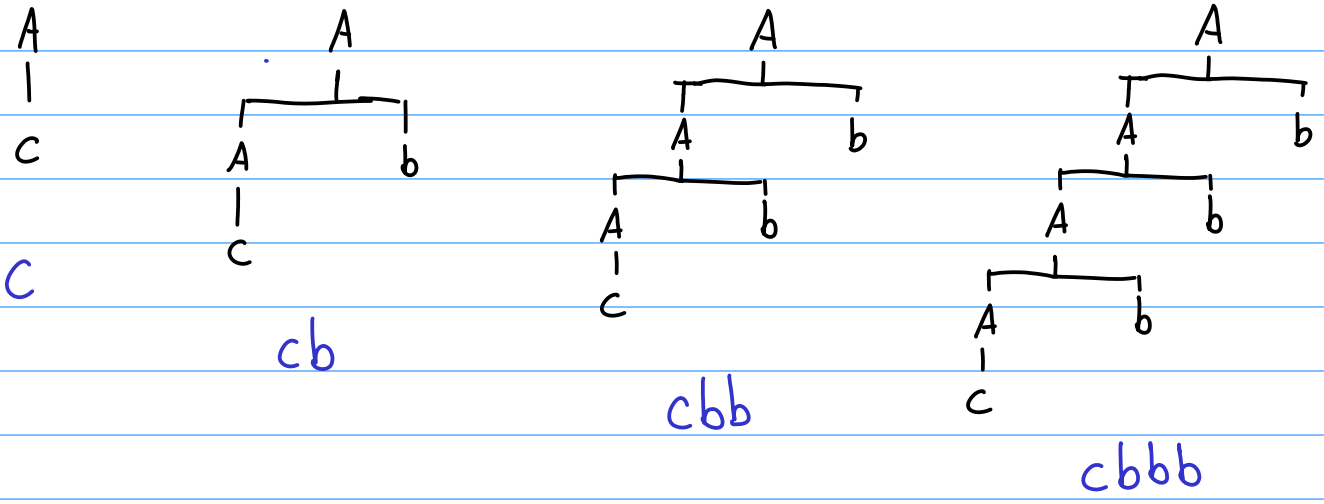
$A \rightarrow Ab$

$A \rightarrow c$

This routine doesn't work

```
void A()
   { if (next_token== '')
       { go_ahead(); }
     else
        { A()
          if (next_token != 'b')
             {Error}
          else
             {go_ahead();}
     }
```

This is a top-down method reading from top to bottom and left to right



c

cb

cbb

cbbb

Formal definition

$\alpha \rightarrow \alpha\beta$

$\alpha \rightarrow \theta$

$\alpha \rightarrow \sigma$

$\Rightarrow$

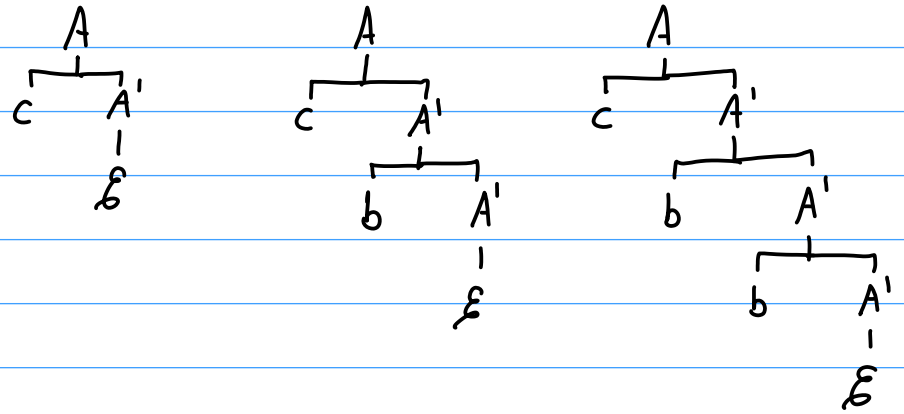
$\alpha \rightarrow \theta\alpha'$

$\alpha \rightarrow \sigma\alpha'$

$\alpha' \rightarrow \beta\alpha'$

$\alpha' \rightarrow \varepsilon$

$A \to Ab$
$A \to c$ $\Rightarrow$ $A \to cA'$
$A' \to \varepsilon$
$A' \to bA'$



The tree is mirrored to the opposite site because ...

Left recursion and Common Left Factor are ambiguities that are cleaned from the grammar

$X \to aY$
$X \to Wb$
$X \to c$
$W \to aW$
$W \to \varepsilon$

The purity of the grammar depends on the tool selected for the lexical and parsing generators, sometimes we need to clean the grammar manually

Operations

Ambiguities $\to$ Common Left Factor / Left Recursion
$\hookrightarrow$ First and Follows calculation

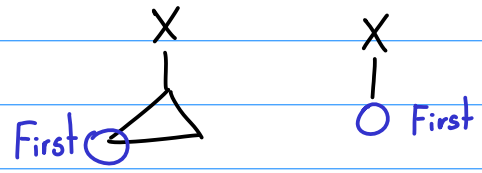
First and Follows are ONLY for variables, not for tokens because they're atomic

First Fi(x)

Set of tokens which can start derivation of a sintactic variable

Tree terminology: set of left-most leafs of all possible trees from the derivation of a sintactic variable

- Every alternative of the variable creates another tree
- We can have null values here



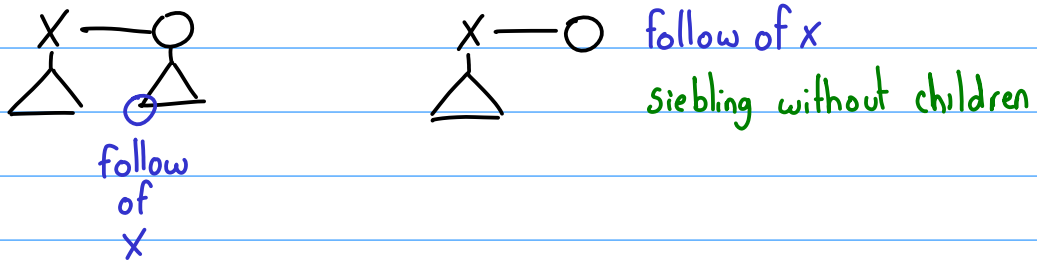
Follows Fo(x)

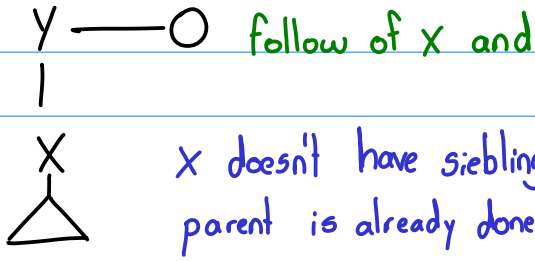
Set of tokens that can appear when finishing the derivation of a sintactic variable

Tree terminology: set of left-most children of the right siebling of a sintactic variable

- We cannot have null values here

- Follows: The initial symbol of the grammar has always the end of file (EOF : $)
- Leafs do not have follows because they're tokens, only variables have Follows





$A \to aBd$ $\quad F_i(A) = \{a, \omega, F_i(B)\} = \{a, \omega, b, f\}$

$A \to \omega$

$A \to Bf$ $\quad F_i(B) = \{b, \varepsilon\}$

$B \to b$

$B \to \varepsilon$ $\quad F_o(A) = \{\$\}$ → end of file $\quad F_o(B) = \{d, f\}$

Traditional grammar

$E \to E + T$ $\quad$ Left recursion ambiguities detected

$E \to T$

$T \to T * F$

$T \to F$

$F \to id$

$F \to (E)$

Applying rules for ambiguities

Left Recursion → $E \to TE'$

$E' \to \varepsilon$

$E' \to +TE'$

$T \to FT'$

$T' \to *FT'$

$T' \to \varepsilon$

$F \to id$

$F \to (E)$

~~~~~~~~~~

$\alpha \to \alpha\beta$

$\alpha \to \theta$

$\alpha \to \theta\alpha'$

$\alpha' \to \varepsilon$

$\alpha' \to \beta\alpha'$

First

$F_i(E) = \{F_i(T)\} = \{id, (\}$

$F_i(E') = \{\varepsilon, +\}$

$F_i(T) = \{F_i(F)\} = \{id, (\}$

$F_i(T') = \{*, \varepsilon\}$

$F_i(F) = \{id, (\}$

Follows

$F_o(E) = \{\$,)\}$

$F_o(T) = \{F_i(E'), F_i(E')\} = \{+, F_o(E'), F_o(E)\}$

$F_o(T') = \{F_o(T), F_o(T')\}$
~~~~~~~~~~

2025/10/13

$Z \to S$
$Z \to \varepsilon$
$S \to Ac$
$S \to Ab$
$A \to xy$
$A \to a$

this grammar has an ambiguity problem

Common Left Factor

$S \to AS'$
$S' \to c$
$S' \to b$
$A \to xy$
$A \to a$

$F_i(Z) = \{F_i(S), \varepsilon\} = \{x, a, \varepsilon\}$
$F_i(S) = \{F_i(A)\} = \{x, a\}$
$F_i(S') = \{c, b\}$
$F_i(A) = \{x, a\}$

$F_o(Z) = \{\$\}$
$F_o(S) = \{\$\}$
$F_o(S') = \{F_o(S)\} = \{\$\}$
$F_o(A) = \{F_i(S')\} = \{c, b\}$

Common Left Factor

$\alpha \to \beta\theta$
$\alpha \to \beta\sigma$
$\alpha \to \mu$

$\alpha \to \mu$
$\alpha \to \beta\alpha'$
$\alpha' \to \theta$
$\alpha' \to \sigma$

Left Recursion

$\alpha \to \alpha\beta$
$\alpha \to \mu$
$\alpha \to \sigma$

$\alpha \to \mu\alpha'$
$\alpha \to \sigma\alpha'$
$\alpha' \to \varepsilon$
$\alpha' \to \beta\alpha'$

This grammar doesn't have any ambiguity at a first glance, but we might find it very deep and hidden after calculating First and Follows

- disjoint sets: we have a problem if we find any disjoint set

$X \to Bwg$
$X \to aBdf$
$X \to cX$
$B \to bBe$
$B \to \varepsilon$

F_i

$F_i(X) = \{a, c, F_i(B)\}$
$= \{a, c, b, \omega\}$
$F_i(B) = \{b, \varepsilon\}$

F_o

they're the same, it is excluded

$F_o(X) = \{\$, F_o(X)\}$
$F_o(B) = \{\omega, d, e\}$

This is the left recursion ambiguity

$E \to E+T$	$E \to TE'$
$E \to T$	$E' \to \varepsilon$
$T \to T*F$	$E' \to +TE'$
$T \to F$	$T \to FT'$
$F \to id$	$T' \to \varepsilon$
$F \to (E)$	$T' \to *FT'$
	$F \to id$ } F does not have any problem, no need to change it
	$F \to (E)$ }

F_i

$F_i(E) = F_i(T)$

$F_i(E') = \{+, \varepsilon\}$

$F_i(T) = \{F_i(F)\} = \{id, c\}$

$F_i(T') = \{*, \varepsilon\}$

$F_i(F) = \{id, c\}$

F_o

$F_o(E) = \{\$,)\}$

$F_o(E') = \{F_o(E), F_o(E')\} = \{\$, c\}$

$F_o(T) = \{F_i(E'), F_i(E')\} = \{+, F_o(E'), F_o(E)\}$
$= \{+, \$, c\}$

no left sibling

$F_o(T') = \{F_o(T), F_o(T')\} = \{+, \$, c\}$

$F_o(F) = \{F_i(T'), F_i(T')\} = \{*, F_o(T), F_o(T')\}$
$= \{*, +, \$, c\}$

Recursive descent for E

```
void E()
{ T();
  E'();
}
```

```
void E();
{ if (next_token != id())
      ERROR
  else
      {
        T();
        E'();
      }
}
```

Function calls are so expensive in terms of memory allocation, runtime

BabyDuck
Entrega 0

Definicion de Gramatica Libre de Contexto

La definición de todas las siguientes gramáticas se realizó con el primer PDF que compartió la profesora, el de Febrero-Junio 2025

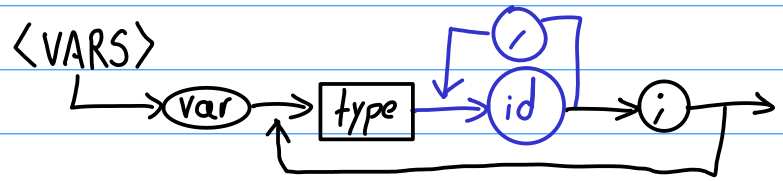
<Programa>

<Programa> → program id ; <OPTIONAL_VARIABLES>
<OPTIONAL_FUNCTIONS> main <Body> end

<OPTIONAL_VARIABLES> → <VARS> | ε

<OPTIONAL_FUNCTIONS> → <FUNCS> | ε

Ejemplo visto en clase de la profa, este diagrama NO coincide con el diagrama de sintaxis del PDF oficial



<VARS> → var <DEC_VAR>

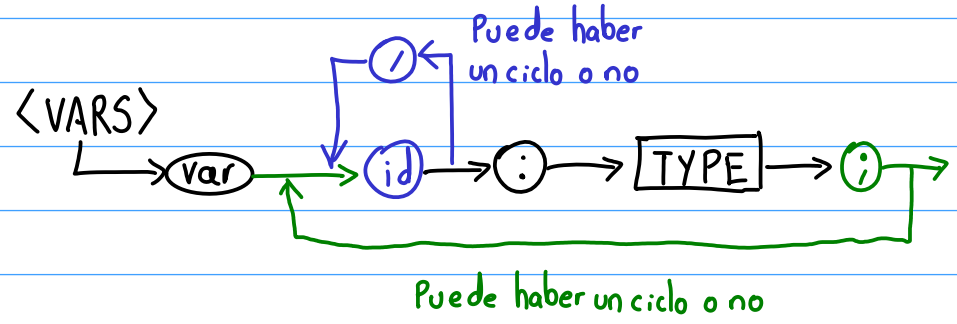
<DEC_VAR> → <TYPE> <LIST_id> ; <DV'>

puede tener 1 o más ids

puede tener un ciclo o no

<DV'> → <DEC_VAR> | ε

Ejemplo exacto del diagrama de sintaxis oficial, la regla gramatical es diferente a la anterior porque producen un lenguaje diferente.



<VARS>

<VARS> → var <DECLARE_VARS>

<DECLARE_VARS> → <DECLARE_IDS> : <TYPE> ; <DECLARE_VARS'>

<DECLARE_VARS'> → <DECLARE_VARS> | ε

<DECLARE_IDS> → id <DECLARE_IDS'>

<DECLARE_IDS'> → , <DECLARE_IDS> | ε

<FUNCS>

<FUNCS> → <RETURN_TYPE> id (<PARAMETERS>)
{ <OPTIONAL_LOCAL_VARIABLES> <Body> } ;

<RETURN_TYPE> → void | <TYPE>

<PARAMETERS> → <PARAMETERS_LIST> | ε

<PARAMETERS_LIST> → id : <TYPE> <MORE_PARAMETERS>

<MORE_PARAMETERS> → , <PARAMETERS_LIST> | ε

<OPTIONAL_LOCAL_VARIABLES> → <VARS> | ε

<Body>
<Body> → { <STATEMENT_LIST> }
<STATEMENT_LIST> → <STATEMENT> <STATEMENT_LIST> | ε

<TYPE>
<TYPE> → int | float

<STATEMENT>
Esta regla presenta una ambigüedad en <ASSIGN> y <F_call> ya que las 2 invocan un id, el primero es para asignar variables y el segundo es para funciones, se resolvió con el factor común izquierdo
<STATEMENT> → id <ID_STATEMENT_CONTINUATION> | <CONDITION>
| <CYCLE> | <Print>
<ID_STATEMENT_CONTINUATION> → = <EXPRESION> ;
| (<EXPRESION_OPCIONAL>) ;
<EXPRESION_OPCIONAL> → <EXPRESION> <MAS_EXPRESIONES> | ε
<MAS_EXPRESIONES> → , <EXPRESION> <MAS_EXPRESIONES> | ε

<ASSIGN>
<ASSIGN> → id = <EXPRESION> ;

<CONDITION>
<CONDITION> → if (<EXPRESION>) <Body>
<OPTIONAL_ELSE> ;
<OPTIONAL_ELSE> → else <Body> | ε

<CYCLE>
<CYCLE> → while (<EXPRESION>) do <Body> ;

<F_Call >
<F_Call> → id (<EXPRESION_OPCIONAL>) ;
<EXPRESION_OPCIONAL> ya fue declarada en <STATEMENT>

<Print>
<Print> → print (<PRINT_LIST>) ;
<PRINT_LIST> → <PRINT_ITEM> <PRINT_MORE_ITEMS>
<PRINT_ITEM> → <EXPRESION> | cte.string
<PRINT_MORE_ITEMS> → , <PRINT_ITEM> <PRINT_MORE_ITEMS> | ε

<EXPRESSION>
<EXPRESSION> → <EXP> <OPTIONAL_COMPARISON>
<OPTIONAL_COMPARISON> → <RELATIONAL_OPERATOR> <EXP> | ε
<RELATIONAL_OPERATOR> → > | < | !=

<EXP>
<EXP> → <TERMINO> <EXP'>
<EXP'> → + <TERMINO> <EXP'> | - <TERMINO> <EXP'> | ε

<TERMINO>
<TERMINO> → <FACTOR> <TERMINO'>
<TERMINO'> → * <FACTOR> <TERMINO'>
| / <FACTOR> <TERMINO'> | ε

<FACTOR>
<FACTOR> → (<EXPRESION>) | <SIN_EXPRESION>
<SIN_EXPRESION> → <SIGN><DATA>
<SIGN> → + | - | ε
<DATA> → id | <CTE>

<CTE>
<CTE> → cte_int | cte_float

BabyDuck
Entrega 0

Gramáticas del PDF 2

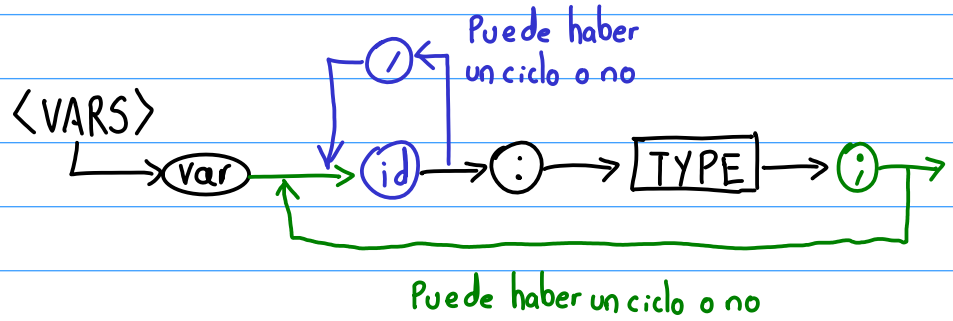
La definición de todas las siguientes gramáticas se realizó con el segundo PDF que compartió la profesora, el de Agosto-Diciembre 2025. Tiene variaciones que producen un lenguaje diferente, además de que las reglas y tokens están escritas en Español, no en Inglés como en el anterior

<Programa>

<Programa> → programa id ; <VARIABLES_OPCIONALES>
<FUNCIONES_OPCIONALES> inicio <CUERPO> fin

<VARIABLES_OPCIONALES> → <VARS> | ε

<FUNCIONES_OPCIONALES> → <FUNCS><FUNCIONES_OPCIONALES> | ε



<VARS>

<VARS> → vars <DECLARAR_VARIABLES>

<DECLARAR_VARIABLES> → <DECLARAR_IDS> : <TIPO> ;
<DECLARAR_VARIABLES'>

<DECLARAR_VARIABLES'> → <DECLARAR_VARIABLES> | ε

<DECLARAR_IDS> → id <DECLARAR_IDS'>

<DECLARAR_IDS'> → , <DECLARAR_IDS> | ε

<FUNCS>

<FUNCS> → <TIPO_DE_RETORNO> id (<PARAMETROS>)
{ <VARIABLES_LOCALES_OPCIONALES> <CUERPO> } ;

<TIPO_DE_RETORNO> → nula | <TIPO>

<PARAMETROS> → <LISTA_DE_PARAMETROS> | ε

<LISTA_DE_PARAMETROS> → id : <TIPO> <MAS_PARAMETROS>

<MAS_PARAMETROS> → , <LISTA_DE_PARAMETROS> | ε

<VARIABLES_LOCALES_OPCIONALES> → <VARS> | ε

<CUERPO>
<CUERPO> → { <LISTA_DE_ESTATUTO> }
<LISTA_DE_ESTATUTO> → <ESTATUTO> <LISTA_DE_ESTATUTO> | ε

<TIPO>
<TIPO> → entero | flotante

<ESTATUTO>
Esta regla presenta una ambigüedad en <ASIGNA> y <LLAMADA> ya que las 2 invocan un id, el primero es para asignar variables y el segundo es para funciones, se resolvió con el factor común izquierdo
<ESTATUTO> → id <CONTINUACION_DE_ESTATUTO_ID> | <CONDICION>
| <CICLO> | <IMPRIME>
| [<ESTATUTO_OPCIONAL>]
<CONTINUACION_DE_ESTATUTO_ID> → = <EXPRESION> ;
| (<EXPRESION_OPCIONAL>) ;
<EXPRESION_OPCIONAL> → <EXPRESION> <MAS_EXPRESIONES> | ε
<MAS_EXPRESIONES> → , <EXPRESION> <MAS_EXPRESIONES> | ε
<ESTATUTO_OPCIONAL> → <ESTATUTO> <ESTATUTO_OPCIONAL> | ε

<ASIGNA>
<ASIGNA> → id = <EXPRESION> ;

<CONDICION>
<CONDICION> → si (<EXPRESION>) <CUERPO>
<SINO_OPCIONAL> ;
<SINO_OPCIONAL> → sino <CUERPO> | ε

<CICLO>
<CICLO> → mientras (<EXPRESION>) haz <CUERPO> ;

<LLAMADA>
<LLAMADA> → id (<EXPRESION_OPCIONAL>)
<EXPRESION_OPCIONAL> ya fue declarada en <ESTATUTO>

<IMPRIME>
<IMPRIME> → escribe (<IMPRIMIR_LISTA>) ;
<IMPRIMIR_LISTA> → <IMPRIMIR_ELEMENTO> <IMPRIMIR_MAS_ELEMENTOS>
<IMPRIMIR_ELEMENTO> → <EXPRESION> | letrero
<IMPRIMIR_MAS_ELEMENTOS> → , <IMPRIMIR_ELEMENTO>
<IMPRIMIR_MAS_ELEMENTOS> | ε

<EXPRESION>
<EXPRESION> → <EXP> <COMPARACION_OPCIONAL>
<COMPARACION_OPCIONAL> → <OPERADOR_RELACIONAL> <EXP> | ε
<OPERADOR_RELACIONAL> → > | < | != | ==

<EXP>
<EXP> → <TERMINO> <EXP'>
<EXP'> → + <TERMINO> <EXP'> | - <TERMINO> <EXP'> | ε

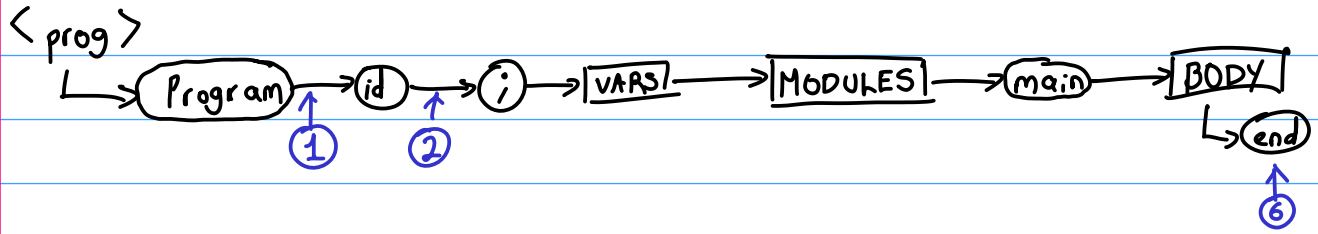
<TERMINO>
<TERMINO> → <FACTOR> <TERMINO'>
<TERMINO'> → * <FACTOR> <TERMINO'>
| / <FACTOR> <TERMINO'> | ε

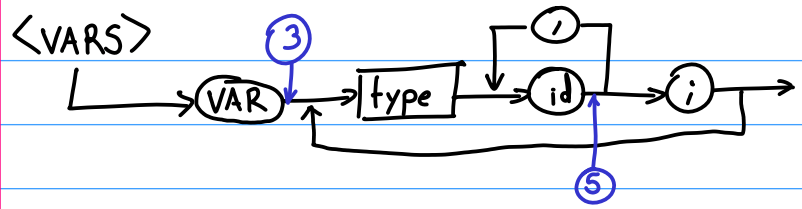
<FACTOR>
<FACTOR> → (<EXPRESION>) | <SIN_EXPRESION>
<SIN_EXPRESION> → <SIGNO> <DATO> | <LLAMADA>
<SIGNO> → + | - | &
<DATO> → id | <CTE>

<CTE>
<CTE> → cte_ent | cte_flot

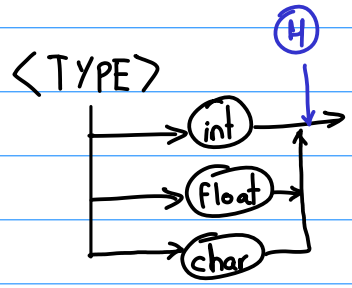
2025/10/16

Insert custom code

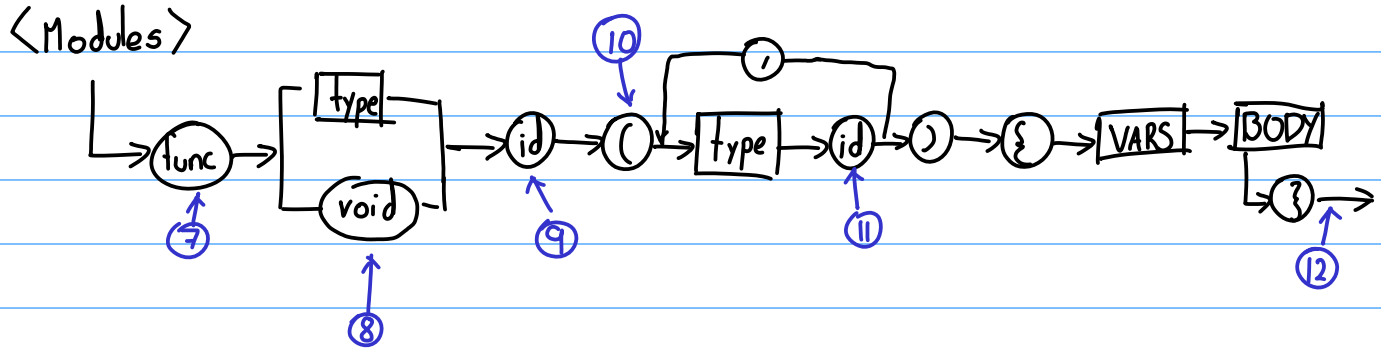




Neuralgic action: stop execution here and define custom rules

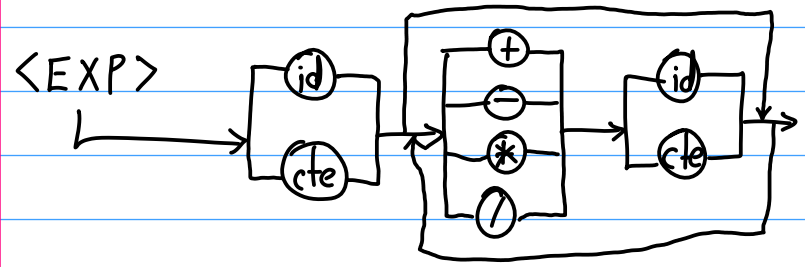


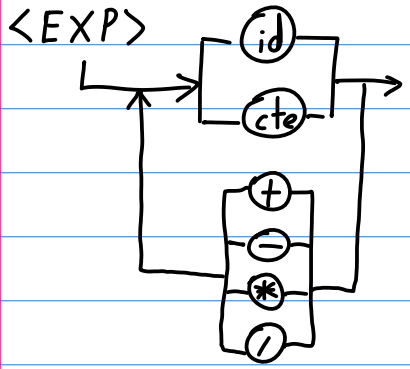
1. Create DirFunc
2. Add id-name and type program a DirFunc
3. If current Func doesn't have a VarTable then
   create VarTable and link it to current Func
4. Current-type = type
5. Search for id-name in current VarTable
   if found → Error "multiple declaration"
   if not, add id-name and current-type to current VarTable
6. Delete DirFunc and current VarTable (Global)



7. Prepare DirFunc to add new function
8. Current-type = void
9. Search for id-name in DirFunc
   if found → Error "multiple declaration"
   if not, add id-name and current-type to it
10. Creata a VarTable and link it to current Func
11. Search for id-name in current VarTable
    if found → Error "Multiple declaration"
    if not, add id-name and current-type to current VarTable
12. Delete current VarTable it's no longer required

2025/10/20



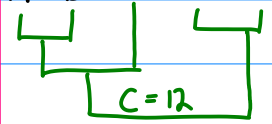


- Both produce A+B, A, A+B*C
- Both expressions are equivalent but not the same
- The syntax only cares that we have the exact sequence

La asociatividad entra en juego cuando no hay jerarquía

Asociativo Izquierdo

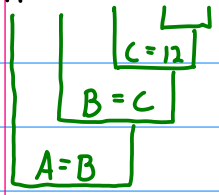
A=B=C=3*4



para que esta operación funcione se debe usar la asociatividad derecha

Asociativo Derecho

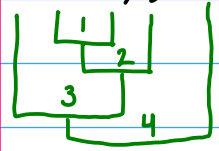
A=B=C=3*4



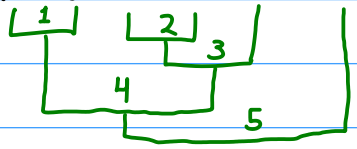
5 AND 4 esta expresión está mal porque solo se aplica con operadores booleanos, a menos que el diseñador del lenguaje lo permita con enteros

Left Associative

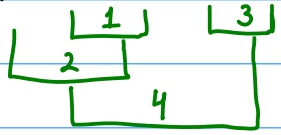
A+B*C/D−E



A*B+C/D*F+G

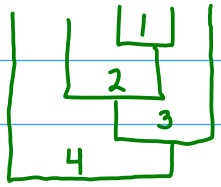


A+B*C−D*E



Right Associative

A+B*C/D-E



CÓDIGO INTERMEDIO

Una representación que no depende de la arquitectura, cada quien decide cómo picar la ceballa.

Técnicas comunes

- Vector Polaco
- Triplos
- Cuádruplos

El código intermedio es libre porque cada quien decide usar su propio algoritmo. Java hizo lo mismo con la JVM al generar bytecode.

Vector Polaco

Una técnica muy sencilla de entender pero que no se recomienda en el proyecto

- Las variables guardan el mismo lugar en la expresión, solo cambia la posición de los símbolos
- Requiere un stack para elementos pendientes y un queue para el resultado
- Las llamadas recursivas requieren la creación de su propia pila y vectores

A + B → AB+

A + B * C → ABC*+

A * B * C →

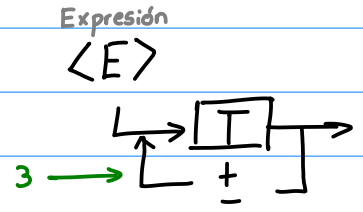
Asociativa Izquierda → AB*C*

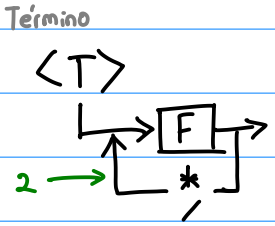
Asociativa Derecha → ~~ABC**~~

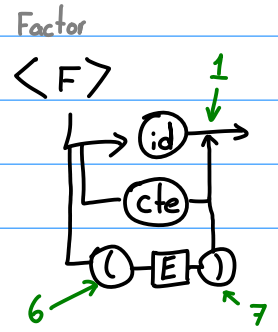
A+B*C/D*E

↳ ABC*D/E*+

Todos los ejercicios siguen los siguientes reglas:







1. vp.push(id|cte)
2. poper.push(*|/)
3. poper.push(+|-)
4. Si: top(poper) == *|/

   ⇒

   op = poper.pop()

   vp.push(op)

5.
6. poper.push('(')
7. poper.pop()

No se pueden tener operadores de la misma prioridad seguidos porque son asociativos izquierdos. El peor escenario es cuando vienen operadores de la misma jerarquía

A + B * C * E - E * F

stack con operadores pendientes (poper)

[+, ~~*~~, ~~*~~, -]

vector polaco (vp)

[A, B, C, *, E, *, +]

A + B - C * D + E / F * G - H

poper

[~~+~~, ~~+~~, ~~*~~, ~~+~~, ~~/~~, ~~*~~, ~~+~~]

vp

[A, B, +, C, D, *, -, E, F, /, G, *, +, H, -]

Cuádruplos

Genera un código más grande que el vector polaco, pero es mucho más fácil de implementar para la máquina virtual.

- Necesitan espacio extra para guardar resultados intermedios
- El orden es en función a la jerarquía del operador, no al orden de las variables. El algoritmo de ejecución es lineal
- Aquí se manejan los tipos de datos con los

$A + B \rightarrow +, A, B, t_1$

$A + B * C \rightarrow *, B, C, t_1$

$+, A, t_1, t_2$

$A + B * C / D \rightarrow *, B, C, t_1$

$/, t_1, D, t_2$

$+, A, t_2, t_3$

$A + B * C / D - E * F$

poper

~~+~~, ~~*~~, ~~/~~]

Pvar's

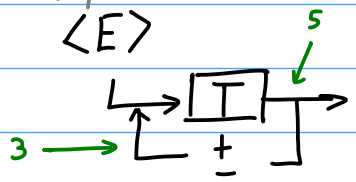
[A, B, C, ~~t_1~~, D, ~~t_2~~, t_3]

cuádruplos

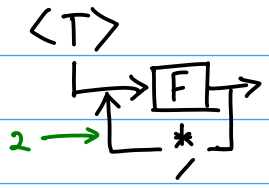
*	B	C	t_1
/	t_1	D	t_2
+	A	t_2	t_3
*	E	F	t_4
-	t_3	t_4	t_5

Todos los ejercicios siguen las siguientes reglas:

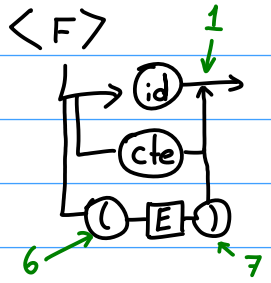
Expresión



Término



Factor



1. Pvar.push(id)
2. poper.push(*||/)
3. poper.push(+|-)
4. Si top(poper)== *||/

   ⇒

   op = poper.pop()

   der = Pvar.pop()

   izq = Pvar.pop()

   t_1 = pedir temporal

A * (B + C / D * (F + G * I) - H * J) + K
int int float int int float float int int float

poper

[* (+ ~~/~~ ~~*~~ ~~(~~ ~~+~~ ~~*~~

Pvars

[A int, B int, ~~C~~ Float, ~~D~~ int, ~~t_1~~ float, ~~F~~ int, ~~G~~ float, ~~I~~ float, ~~t_2~~ float, ~~t_3~~ float, t_4 float

Cuádruplos

/ C D t_1

+ F t_{2F} t_{3F}

* t_{1F} t_{3F} t_{4F}

Estatutos Lineales

Asignación

Escritura

write (A+B, C*D, E)

<ESCRIBE>

Lectura

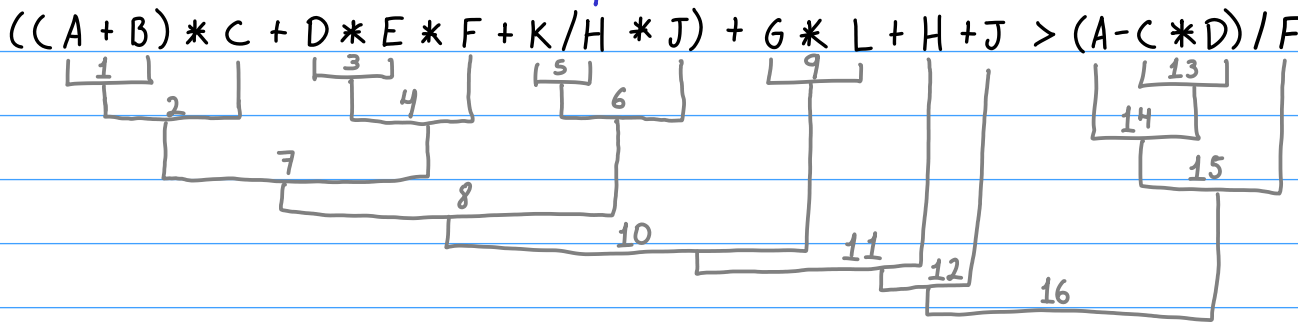
Se refiere a un input, scanf, cin>>

La semántica de la lectura dice que se debe mandar una variable, no un id

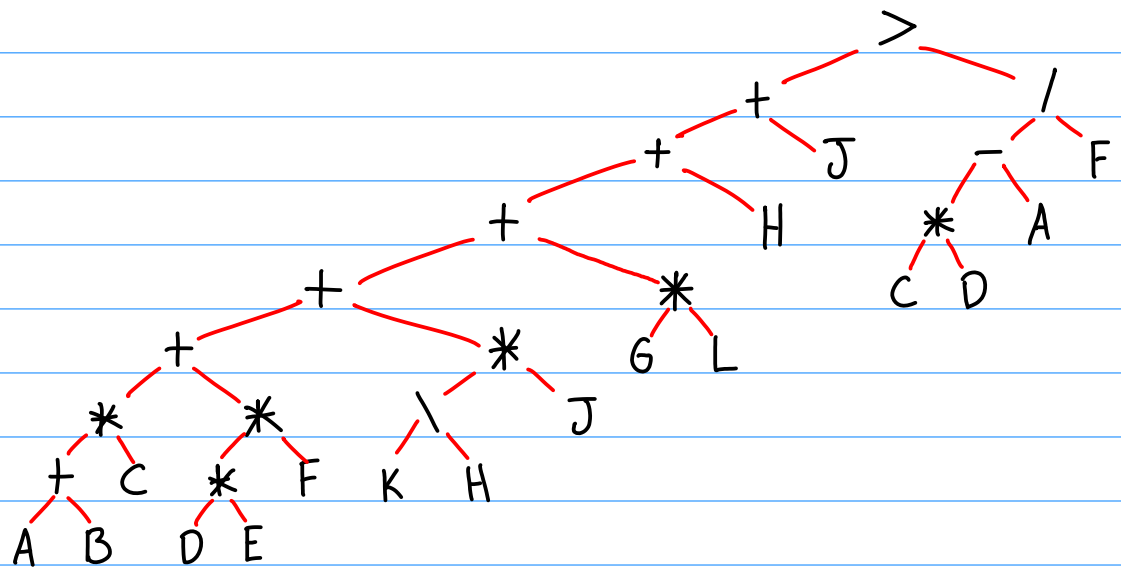
2025/11/04 Tarea 3. Expresiones

((A + B) * C + D * E * F + K / H * J) + G * L + H + J > (A - C * D) / F

Árbol de evaluación asociativo izquierdo

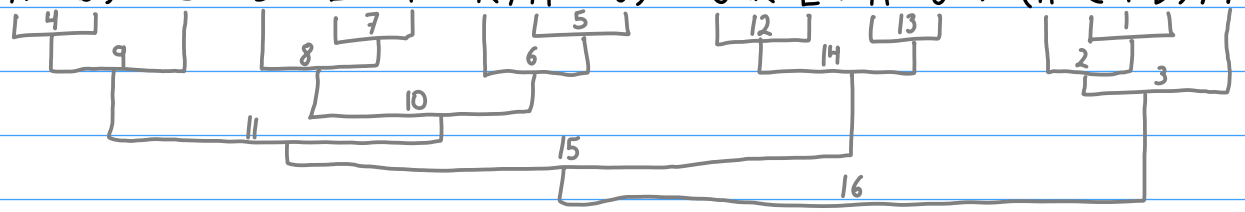


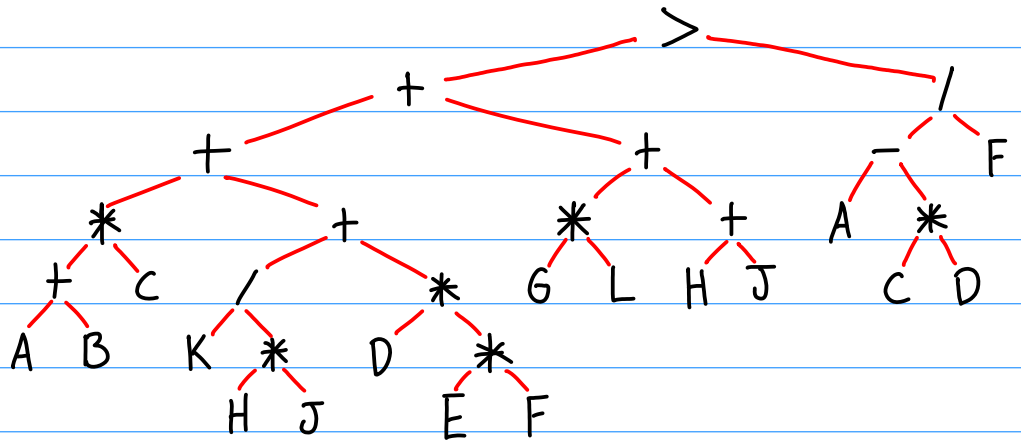
El árbol se construye con un recorrido post-order, la raíz y los nodos cercanos a ella son los operadores con menor precedencia mientras que las hojas y los nodos cercanos a ellas contienen los operadores de mayor precedencia



Árbol de evaluación asociativo derecho

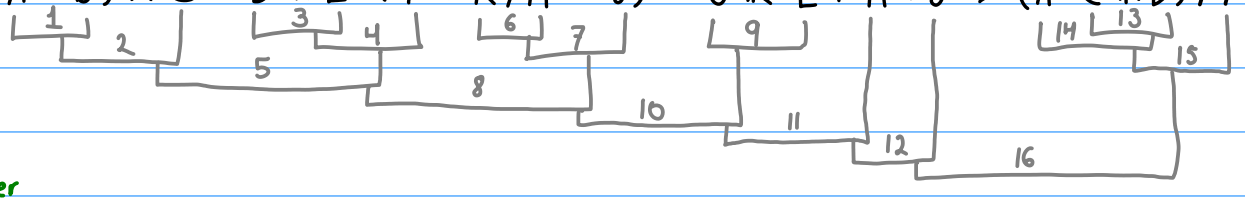
((A + B) * C + D * E * F + K / H * J) + G * L + H + J > (A - C * D) / F





Vector Polaco asociativo izquierdo

((A + B) * C + D * E * F + K / H * J) + G * L + H + J > (A - C * D) / F



poper

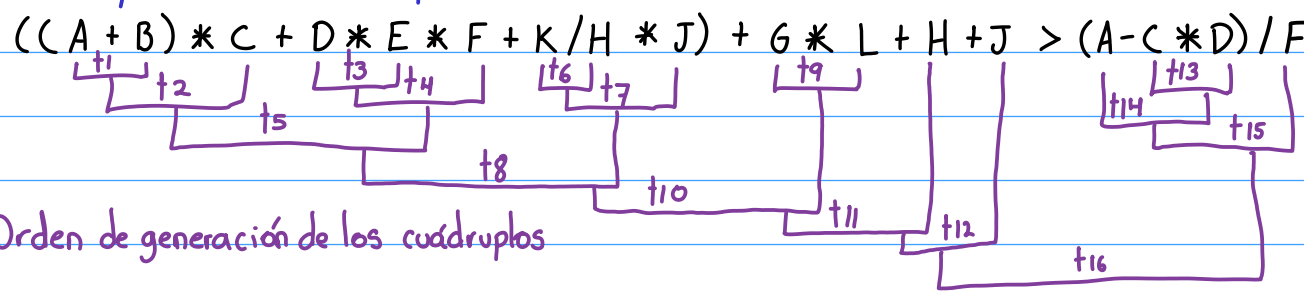
[((+) * + * * + / *) + * + + > (- *) /]

1 2 5 3 4 8 6 7 10 9 11 12 16 14 13 15

vp

[A B + C * D E * F * + K H / J * + G L * + H + J + A C D * - F >]

Cuádruplos asociativo izquierdo



Orden de generación de los cuádruplos

poper

[((A + B) * C + D * E * F + K / H * J) + G * L + H + J > (A - C * D) / F]

t1 t2 t5 t3 t4 t8 t6 t7 t10 t9 t11 t12 t16 t14 t13 t15

Pvar's

[A B t1 C t2 D E t3 F t4 t5 K H t6 J t7 t8 G L t9 t10 H t11 J t12
A C D t13 t14 F t15 t16]

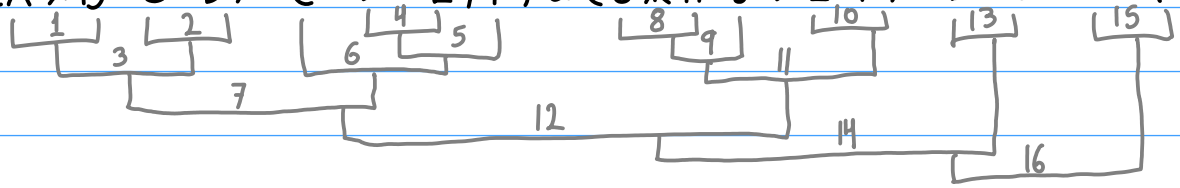
Cuádruplos

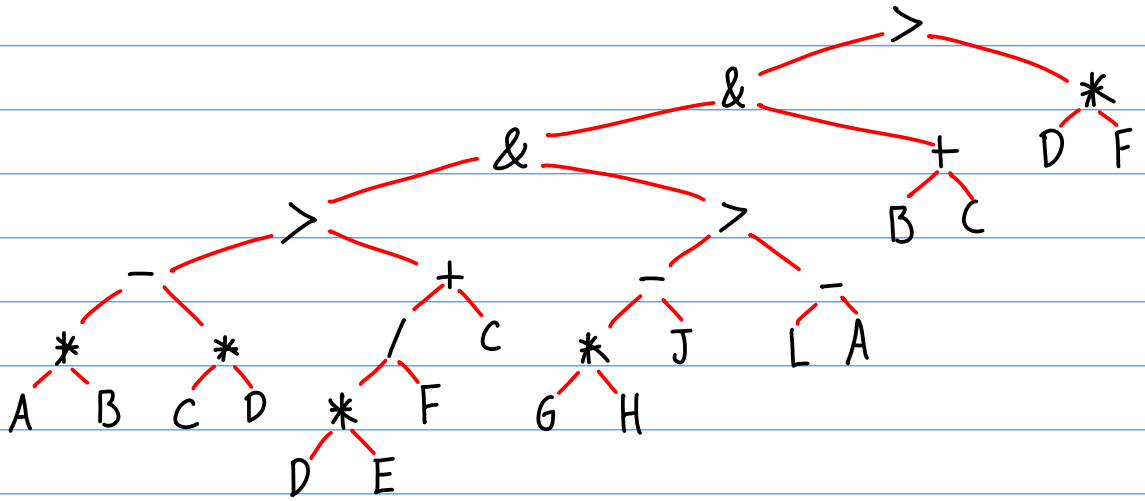
+ A B t1
* t1 C t2
* D E t3
* t3 F t4
+ t2 t4 t5
/ K H t6
* t6 J t7
+ t5 t7 t8
* G L t9
+ t8 t9 t10
+ t10 H t11
+ t11 J t12
* C D t13
- A t13 t14
/ t14 F t15
> t12 t15 t16

((A*B-C*D)>C+D*E/F)&(G*H-J>L-A)&B+C>D*F

Árbol de evaluación asociativo izquierdo

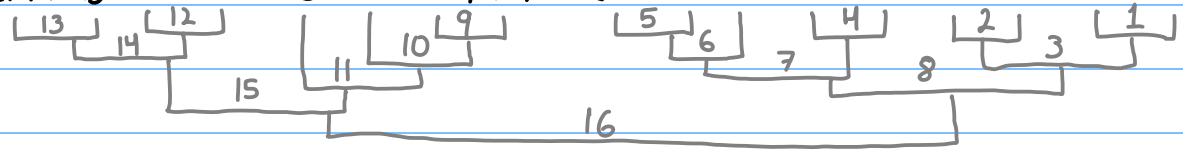
((A*B-C*D)>C+D*E/F)&(G*H-J>L-A)&B+C>D*F

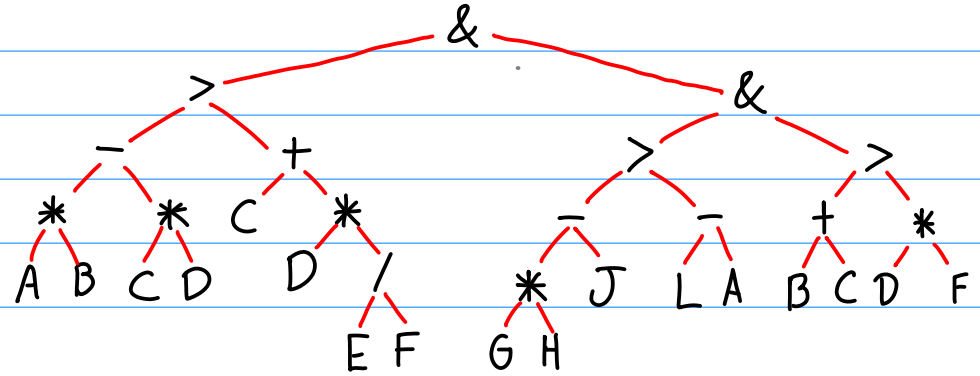




Árbol de evaluación asociativo derecho

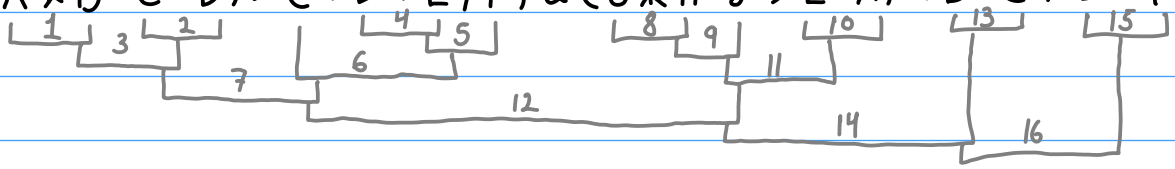
((A*B-C*D)>C+D*E/F)&(G*H-J>L-A)&B+C>D*F





Vector Polaco asociativo izquierdo

((A*B-C*D)>C+D*E/F)&(G*H-J>L-A)&B+C>D*F



poper

[((A B * C D * - C D E * F / + > & (G H * J - L A - > & B C + D F * >]

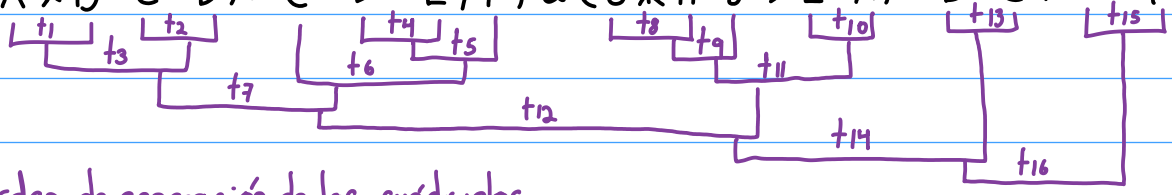
1 3 2 7 6 4 5 12 8 9 11 10 14 13 16 15

vp

[A B * C D * - C D E * F + > G H * J - L A - > & B C + & D F * >]

Cuádruplos asociativo izquierdo

$((A*B-C*D)>C+D*E/F)\&(G*H-J>L-A)\&B+C>D*F$



Orden de generación de los cuádruplos

poper

[((* + *) > + * /) & (* + > +) & + > *]

t1 t3 t2 t7 t6 t4 t5 t12 t8 t9 t11 t10 t14 t13 t16 t15

Prar's

[A B t1 C D t2 t3 C D E t4 F t5 t6 t7 G H t8 J t9 L A t10 t11 t12
B C t13 t14 D F t15 t16]

Cuádruplos

*	A	B	t_1
*	C	D	t_2
-	t_1	t_2	t_3
*	D	E	t_4
/	t_4	F	t_5
+	C	t_5	t_6
>	t_3	t_6	t_7
*	G	H	t_8
-	t_8	J	t_9
-	L	A	t_{10}
>	t_9	t_{10}	t_{11}
&	t_7	t_{11}	t_{12}
+	B	C	t_{13}
&	t_{12}	t_{13}	t_{14}
*	D	F	t_{15}
>	t_{14}	t_{15}	t_{16}

2025/11/10

```
// si la condición es falsa la VM no sabe que tan grande es el salto, se trata de una
// ejecución no lineal
if(A+B > C*D){
    A=B+C;
}
B=D+E;
```

Esto se puede analizar desde la ejecución sintáctica y el flujo de ejecución

```
+ A B t1
* C D t2
> t1 t2 t3
GOTOF          No ejecutar los cuádruplos de TRUE si la condición es falsa
+ B C t4
= t4 A
+ D E t5
= t5 B
```

Comportamiento de un if

- Validar cuádruplo de toda la condición
- Saltar con GOTOF al cuádruplo afuera del estatuto if si la condición es False
- El siguiente cuádruplo es del contenido del estatuto if si la condición es True. Continuar hasta leer todo el contenido del if
- El siguiente cuádruplo corresponde al contenido afuera del estatuto if

Comportamiento de un if-else

- Validar cuádruplo de toda la condición
- Saltar al número de cuádruplo del else con un GOTOF y analizar directamente los cuádruplos del else. Al finalizar los cuádruplos del else, empieza el análisis de los cuádruplos afuera del estatuto if-else
- Continuar con los cuádruplos normales cuando la condición es True. Al terminar los cuádruplos saltar con un GOTO al siguiente cuádruplo del final del else

NOTA: al tener varios if-else anidados y al analizar una expresión muy interna ya sea un if o un else, hay que saltar al final del estatuto que engloba a la línea actual

Comportamiento de un while

1. Validar cuádruplo de toda la condición
2. Saltar al cuádruplo que sigue afuera del while con GOTOF si la condición es False
3. Analizar todos los cuádruplos del while si la condición es True
4. Repetir el paso 1: saltar con un GOTO al primer cuádruplo que evalúa la condición del while

// Se usa un stack para manejar casos pendientes

```
if(A+B > C*D){
    A=B+C;
}
else{
    C=D-A;
    print(c);
}
B=D+E;
```

Stack:

Guarda las posiciones de los saltos goto

[~~4~~, ~~7~~]

1.	+	A	B	t_1
2.	*	C	D	t_2
3.	>	t_1	t_2	t_3
4.	gotoF	t_3	8	
5.	+	B	C	t_4
6.	=	t_4		A
7.	goto	11		
8.	-	D	A	t_5
9.	=	t_5		C
10.	print			c
11.	+	D	E	t_6
12.	=	t_6		B
13.				

Si t_3 es False moverse al cuádruplo 8, en caso contrario continuar desde el cuádruplo 5 en adelante

Stack de saltos: [$, ~~8~~, ~~9~~, ~~11~~, ~~18~~, ~~21~~, ~~22~~, ~~23~~]
(if 1, if 2, while 1, else 2, else 1, while 2)

```
if (A * B - C > D + E) {
    if (B > C + D) {
        while (A + B > D) {
            A = A - C;
            print(A, B + C);
        }
    }
    else { B = C + D; }
}
else {
    while (A > B) {
        B = B - C;
        print(C);
    }
}
B = C * D;
A = B + C * D;
```

1.	*	A	B	t_1	
2.	-	t_1	C	t_2	
3.	+	D	E	t_3	
4.	>	t_2	t_3	t_4	Expresión de if 1
5.	gotoF	t_4		22	Ir a else 1
6.	+	C	D	t_5	if 1 es True
7.	>	B	t_5	t_6	Expresión de if 2
8.	gotoF	t_6		19	Ir a else 2
9.	+	A	B	t_7	If 2 es True → Evaluar while 1
10.	>	t_7	D	t_8	
11.	gotoF	t_8		18	Expresión de while 1 es False
12.	-	A	C	t_9	Expresión de while 1 es True
13.	=	t_9		A	
14.	print			A	
15.	+	B	C	t_{10}	
16.	print			t_{10}	
17.	goto	9			Re-evaluar while 1 →
18.	goto	21			fin del if 2, saltar al final de la sentencia más reciente (if 1) →
19.	+	C	D	t_{11}	
20.	=	t_{11}		B	
21.	goto	28			fin del if 1 salir
22.	>	A	B	t_{12}	Evaluar while 2 →
23.	gotoF	t_{12}		28	Expresión de while 2 es False →
24.	-	B	C	t_{13}	Expresión de while 2 es True →
25.	=	t_{13}		B	
26.	print			C	
27.	goto	22			Re-evaluar while 2 →
28.	*	C	D	t_{14}	
29.	=	t_{14}		B	
30.	*	C	D	t_{15}	
31.	+	B	t_{15}	t_{16}	
32.	=	t_{16}		A	
33.					
34.					
35.					

Análisis de invarianza:

2025/11/14 **Tarea 4: Estatutos. Generación de Código Intermedio**

Genera las acciones de código intermedio basado en cuádruplos para el siguiente segmento de código:

```
A = B + C * (D - E / F) * H;
B = E - F;
WHILE  A * B - C >= D * E / (G + H) {
    H = J * K + B;
    IF (B < H) {
        B = H + J
        WHILE (B > A + C) {
            PRINT(A + B * C, D - E);
            B = B - J;
        }
    }
    ELSE {
        DO {
            A = A + B;
            PRINT(B - D);
        }
        WHILE (A - D < C + B);
    }
}
F = A + B;
```

stack de saltos = [15 (while 1), ~~20~~ (else 1), ~~25~~ (while 2), ~~34~~ (Fin del if 1), ~~35~~ (DO)]

1.	/	E	F	t_1	
2.	-	D	t_1	t_2	
3.	*	C	t_2	t_3	
4.	*	t_3	H	t_4	
5.	+	B	t_4	t_5	
6.	=	t_5		A	
7.	-	E	F	t_6	
8.	=	t_6		B	
9.	*	A	B	t_7	Evaluar WHILE 1
10.	-	t_7	C	t_8	
11.	+	G	H	t_9	
12.	*	D	E	t_{10}	
13.	/	t_{10}	t_9	t_{11}	
14.	>=	t_8	t_{11}	t_{12}	
15.	GOTOF	t_{12}	45		Terminar WHILE 1
16.	*	J	K	t_{13}	Continuar WHILE 1
17.	+	t_{13}	B	t_{14}	
18.	=	t_{14}		H	
19.	<	B	H	t_{15}	IF 1 expresión
20.	GOTOF	t_{15}	35		ELSE 1
21.	+	H	J	t_{16}	IF 1 EXP = True
22.	=	t_{16}		B	
23.	+	A	C	t_{17}	Evaluar WHILE 2
24.	>	B	t_{17}	t_{18}	
25.	GOTOF	t_{18}	34		Terminar WHILE 2
26.	*	B	C	t_{19}	Continuar WHILE 2

27. + A t_{19} t_{20}

28. PRINT t_{20}

29. - D E t_{21}

30. PRINT t_{21}

31. - B J t_{22}

32. = t_{22} B

Re-evaluar WHILE 2

33. GOTO 23

Fin del if 1

34. GOTO 44

Primer cuádruplo del DO-WHILE

35. + A B t_{23}

36. = t_{23} A

37. - B D t_{24}

38. PRINT t_{24}

Evaluar WHILE de DO-WHILE

39. - A D t_{25}

40. + C B t_{26}

41. < t_{25} t_{26} t_{27}

Repetir lógica del DO

42. GOTOV t_{27} 35

Salir del DO-WHILE

43. GOTO 44

Fin del IF-ELSE 1

Re-evaluar WHILE 1

44. GOTO 9

45. + A B t_{28}

46. = t_{28} F