

Compiladores

2025/09/29

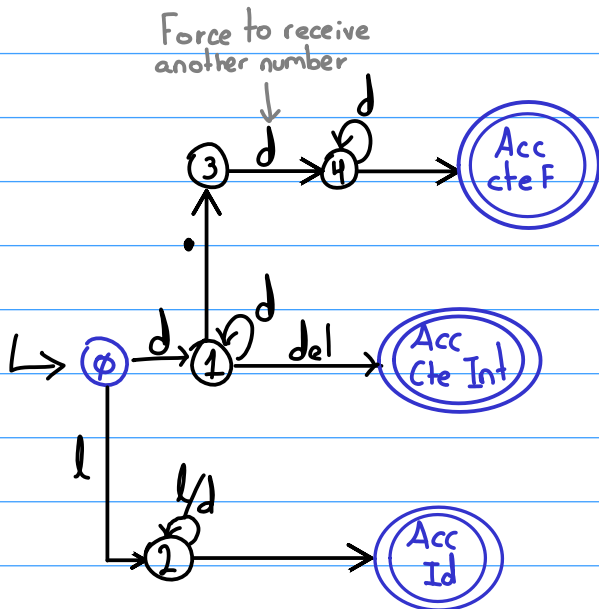
white space



Let $d = \{0, \dots, 9\}$

let $l = \{a, \dots, z\}$

$del = \{ \text{space}, \text{ret}, +, *, \dots, \}$



2025/09/29

Defining the sets helps reducing the complexity of the matrix

1235 \rightarrow dec ✓

0101B \rightarrow bin ✓

0101 \rightarrow dec ✓

35BEh \rightarrow hex ✓

ØABH \rightarrow hex ✓

ABH \rightarrow ID

float ✓

sets $d = \{0, \dots, 9\}$

$d_bin = \{0, 1\}$

$no_bin = \{2, \dots, 9\}$

$\mathcal{L} = \{a, \dots, z, A, \dots, Z\}$

$bin = \{B, b\}$

$Hex = \{H, h\}$

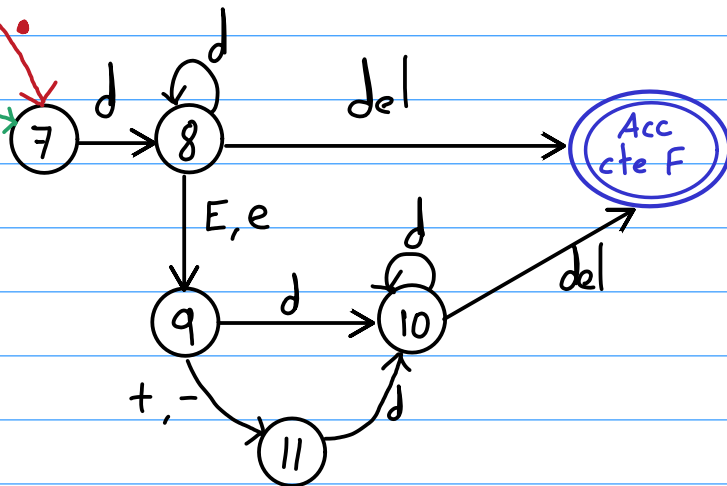
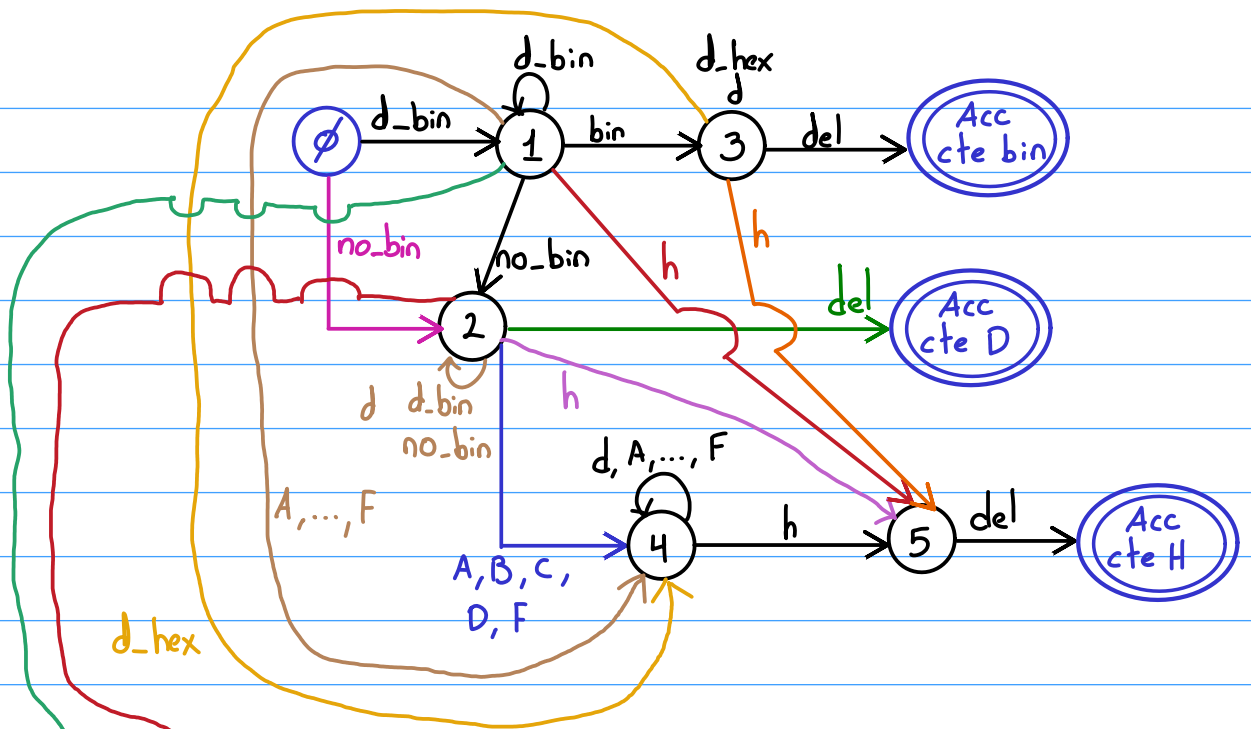
$d_hex = \{A, C, D, E, F\} + bin + d - Bb - Ee$

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 syntax 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



Syntax diagrams

Notation

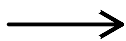


tokens / terminal symbols



syntactic variables / non-terminal symbols

this is a way to summarize modules



flux (direction), this is unidirectional

<rule>

rule development



Example

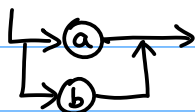
<x>

$L = \{a\}$



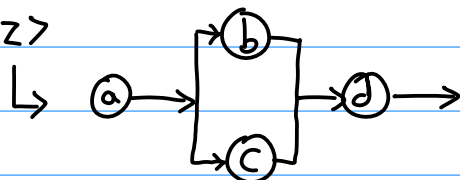
<y>

$L = \{a, b\}$

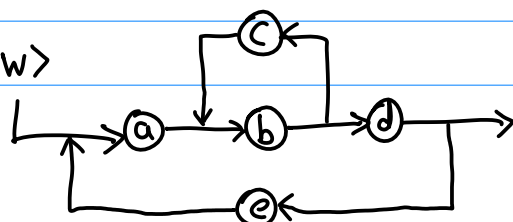


<z>

$L = \{abd, acd\}$

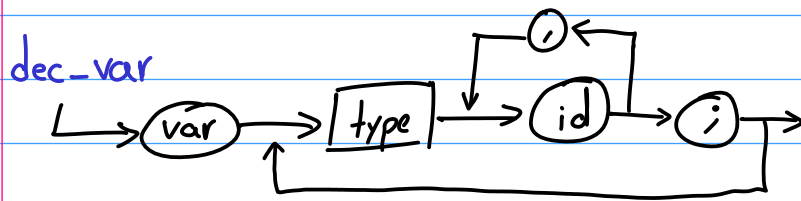


<w>



... incomplete ...

2 different styles to declare variables



dec_var(2)

Formal Grammars

$\alpha \rightarrow \beta$ alpha produces beta

$E \rightarrow E + E$

$A = \dots C * D$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

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

$E \Rightarrow E + E$

$\Rightarrow id + E$

$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$ ✓

$E \Rightarrow E * E$

$\Rightarrow E + E * E$

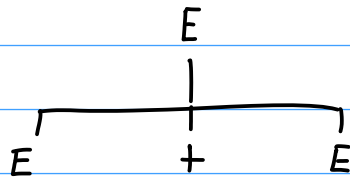
$\Rightarrow id + E * E$

\dots

$\Rightarrow id + id * id$ ✓

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 (\emptyset)

Z : capital Z

z : lowercase Z

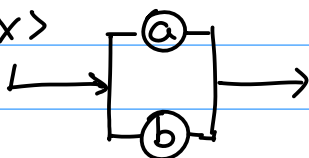
\$: end of file

$\langle x \rangle$



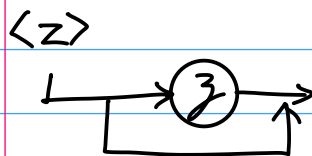
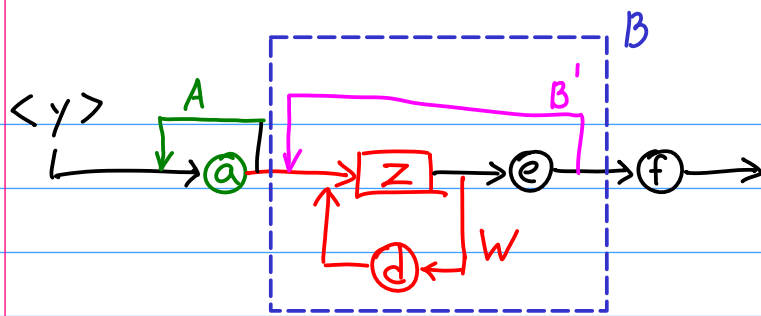
$x \rightarrow a$ read as "x produces a"

$\langle x \rangle$



$X \rightarrow a$

$x \rightarrow b$



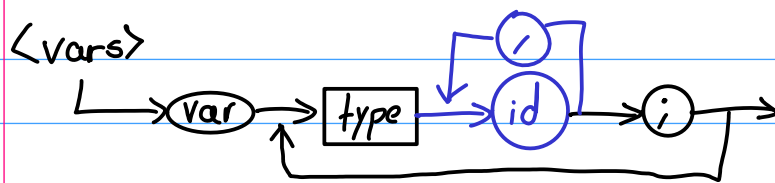
$z \rightarrow z$
 $z \rightarrow \epsilon$

Z produces z (lowercase z) or
 nothing

$A \rightarrow a$
 $A \rightarrow aA \equiv A' \rightarrow \epsilon$
 $A' \rightarrow \langle A \rangle$

$B \rightarrow weB'$
 $B' \rightarrow \epsilon$
 $B' \rightarrow B$
 $W \rightarrow \langle z, w' \rangle$
 $w' \rightarrow$

Both expressions are equivalent,
 but the right option is better



$\langle \text{Vars} \rangle \rightarrow \text{var} \langle \text{DEC_VAR} \rangle$

$\langle \text{DEC_VAR} \rangle \rightarrow \langle \text{TYPE} \rangle \langle \text{LIST_id} \rangle ; \langle \text{DV}' \rangle$

could be 1 or more ids could cycle or not

$\langle \text{DV}' \rangle \rightarrow \epsilon$

$\langle \text{DV}' \rangle \rightarrow \langle \text{DEC_VAR} \rangle$

2025/10/09

Brute-Force method

It supposes that every syntactic variable...

Another module involves a lexicon

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 linear method because it uses a brute-force approach

Example

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

$A \rightarrow aB$

$A \rightarrow c$

↓

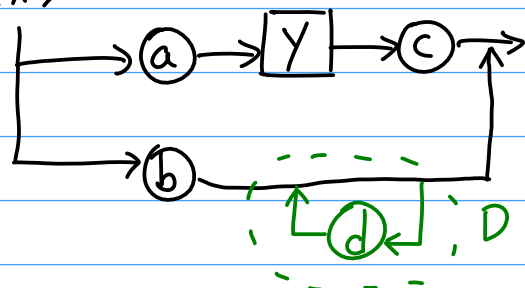
void A()

{ if

Example

1st step: get grammar rules

$\langle x \rangle$



$X \rightarrow aYc$

$X \rightarrow bD$

$D \rightarrow \epsilon$

$D \rightarrow dD$

$\langle y \rangle$



$Y \rightarrow w$

$Y \rightarrow \epsilon$

Programming function $Y()$

void $Y()$

```
{ if (next_token == 'w') // 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();
```

```
    }
```

```
  }
```

```
    Error(a,b) // you have either a or b, not anything else
```

$$X \rightarrow aYb$$
$$X \rightarrow dw$$
$$X \rightarrow aB$$

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

$$X \rightarrow dw$$
$$X \rightarrow aX'$$

X' is the uncommon part

$$X' \rightarrow Yb$$
$$X' \rightarrow B$$

Common Left Factor

$$\alpha \rightarrow \beta\theta$$
$$\alpha \rightarrow w$$
$$\alpha \rightarrow \beta\sigma \Rightarrow \alpha \rightarrow \beta\alpha'$$
$$\alpha \rightarrow w \quad \alpha' \rightarrow \theta$$
$$\alpha' \rightarrow \sigma$$

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

A
|
c

c

A
├── A
│ |
│ c
└── b

cb

A
├── A
│ ├── A
│ │ |
│ │ c
│ └── b
└── b

cbb

A
├── A
│ ├── A
│ │ ├── A
│ │ │ |
│ │ │ c
│ │ └── b
│ └── b
└── b

cbbb

Formal definition

$\alpha \rightarrow \alpha\beta$

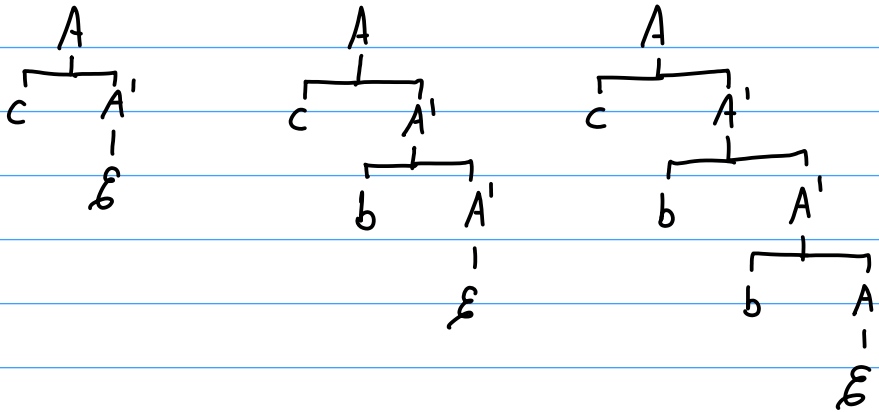
$\alpha \rightarrow \theta\alpha'$

$\alpha \rightarrow \theta \Rightarrow \alpha \rightarrow \sigma\alpha'$

$\alpha \rightarrow \sigma \quad \alpha' \rightarrow \beta\alpha'$

$\alpha' \rightarrow \epsilon$

$$\begin{aligned}
 A &\rightarrow Ab & A &\rightarrow cA' \\
 A &\rightarrow c & \Rightarrow & A' \rightarrow \epsilon \\
 & & & A' \rightarrow bA'
 \end{aligned}$$



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

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

$$\begin{aligned}
 X &\rightarrow \underline{a}Y \\
 X &\rightarrow Wb \\
 X &\rightarrow c \\
 W &\rightarrow \underline{a}W \\
 W &\rightarrow \epsilon
 \end{aligned}$$

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 \rightarrow Common Left Factor / Left Recursion
 \rightarrow First and Follows calculation

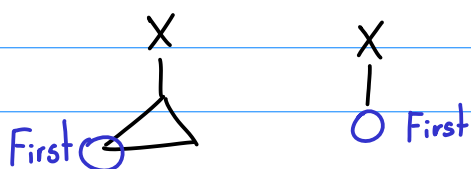
First and **Follows** are ONLY for variables, not for tokens because they're atomic

First $F_i(x)$

Set of tokens which can start derivation of a syntactic variable

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

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



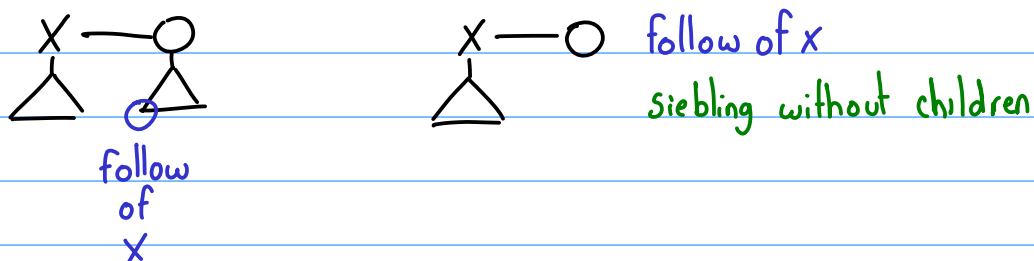
Follows $F_o(x)$

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

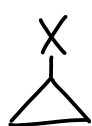
Tree terminology: set of left-most children of the right sibling of a syntactic variable

- 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



Y — \bigcirc follow of x and y : $F_o(Y)$, $F_o(X)$



x doesn't have siblings, after finishing its derivation, its closest parent is already done, and is done the closest follow

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

$$A \rightarrow w$$

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

$$B \rightarrow b$$

$$B \rightarrow \varepsilon$$

$$F_o(A) = \{ \$ \}$$

→ end of file

$$F_o(B) = \{d, f\}$$

Traditional grammar

$$E \rightarrow E + T$$

Left recursion ambiguities detected

$$E \rightarrow T$$

$$T \rightarrow T * F$$

Applying rules for ambiguities

$$T \rightarrow F$$

Left Recursion $\rightarrow E \rightarrow TE'$

$$F \rightarrow id$$

$$E' \rightarrow \varepsilon$$

$$F \rightarrow (E)$$

$$E' \rightarrow +TE'$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT'$$

$$T' \rightarrow \varepsilon$$

$$F \rightarrow id$$

$$F \rightarrow (E)$$

$$\alpha \rightarrow \alpha B$$

$$\alpha \rightarrow \theta$$

$$\alpha \rightarrow \theta \alpha'$$

$$\alpha' \rightarrow \varepsilon$$

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

First

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

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

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

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

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

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

$Z \rightarrow S$

this grammar has an ambiguity problem

$Z \rightarrow \epsilon$

$$F_i(Z) = \{F_i(S), \epsilon\} = \{x, a, \epsilon\}$$

$S \rightarrow Ac$ } Common Left Factor } $S \rightarrow AS'$

$$F_i(S) = \{F_i(A)\} = \{x, a\}$$

$S \rightarrow Ab$ } $S' \rightarrow c$

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

$A \rightarrow xy$ $S' \rightarrow b$

$$F_i(A) = \{x, a\}$$

$A \rightarrow a$ $A \rightarrow xy$

$A \rightarrow a$

$$F_0(Z) = \{\$ \}$$

Common Left Factor

Left Recursion

$$F_0(S) = \{\$ \}$$

$$\alpha \rightarrow \beta \theta$$

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

$$F_0(S') = \{F_0(S)\} = \{\$ \}$$

$$\alpha \rightarrow \beta \sigma$$

$$\alpha \rightarrow \mu$$

$$F_0(A) = \{F_i(S')\} = \{c, b\}$$

$$\alpha \rightarrow \mu$$

$$\alpha \rightarrow \sigma$$

$$\alpha \rightarrow \mu$$

$$\alpha \rightarrow \mu \alpha'$$

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

$$\alpha \rightarrow \sigma \alpha'$$

$$\alpha' \rightarrow \theta$$

$$\alpha' \rightarrow \epsilon$$

$$\alpha' \rightarrow \sigma$$

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

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

$X \rightarrow Bwg$

F_i

F_0

they're the same, it is excluded

$X \rightarrow aBdf$

$$F_i(x) = \{a, c, F_i(B)\}$$

$$F_0(x) = \{\$, F_0(x)\}$$

$X \rightarrow cX$

$$= \{a, c, b, w\}$$

$$F_0(B) = \{w, d, e\}$$

$B \rightarrow bBe$

$$F_i(B) = \{b, \epsilon\}$$

$B \rightarrow \epsilon$

This is the left recursion ambiguity

$$E \rightarrow E + T$$

$$E \rightarrow TE'$$

$$E \rightarrow T$$

$$E' \rightarrow \epsilon$$

$$T \rightarrow T * F$$

$$E' \rightarrow +TE'$$

$$T \rightarrow F$$

$$T \rightarrow FT'$$

$$F \rightarrow id$$

$$T' \rightarrow \epsilon$$

$$F \rightarrow (E)$$

$$T' \rightarrow *FT'$$

$$F \rightarrow id$$

$$F \rightarrow (E)$$

} F does not have any problem, no need to change it

F_i

$$F_i(E) = F_i(T)$$

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

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

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

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

F_o

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

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

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

$$= \{+, \$, c\}$$

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

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

$$= \{*, +, \$, c\}$$

no left sibling

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

Definición de Gramática 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

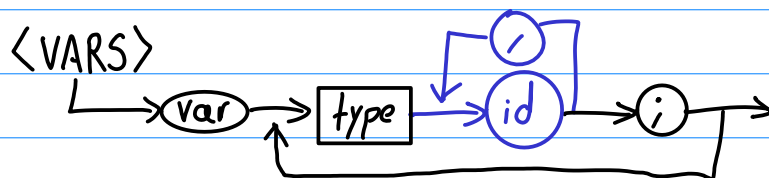
$\langle \text{Programa} \rangle$

$\langle \text{Programa} \rangle \rightarrow \text{program id ; } \langle \text{OPTIONAL_VARIABLES} \rangle$
 $\langle \text{OPTIONAL_FUNCTIONS} \rangle \text{ main } \langle \text{Body} \rangle \text{ end}$

$\langle \text{OPTIONAL_VARIABLES} \rangle \rightarrow \langle \text{VARS} \rangle \mid \epsilon$

$\langle \text{OPTIONAL_FUNCTIONS} \rangle \rightarrow \langle \text{FUNCS} \rangle \mid \epsilon$

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



$\langle \text{VARS} \rangle \rightarrow \text{var } \langle \text{DEC_VAR} \rangle$

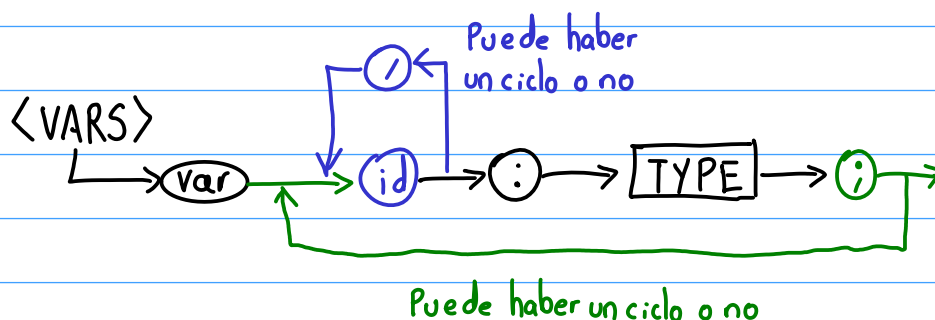
$\langle \text{DEC_VAR} \rangle \rightarrow \langle \text{TYPE} \rangle \langle \text{LIST_id} \rangle ; \langle \text{DV}' \rangle$

puede tener 1
o más ids

puede tener un
ciclo o no

$\langle \text{DV}' \rangle \rightarrow \langle \text{DEC_VAR} \rangle \mid \epsilon$

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

$\langle \text{Body} \rangle$

$\langle \text{Body} \rangle \rightarrow \{ \langle \text{STATEMENT_LIST} \rangle \}$

$\langle \text{STATEMENT_LIST} \rangle \rightarrow \langle \text{STATEMENT} \rangle \langle \text{STATEMENT_LIST} \rangle \mid \epsilon$

$\langle \text{TYPE} \rangle$

$\langle \text{TYPE} \rangle \rightarrow \text{int} \mid \text{float}$

$\langle \text{STATEMENT} \rangle$

Esta regla presenta una ambigüedad en $\langle \text{ASSIGN} \rangle$ y $\langle \text{F-call} \rangle$ 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

$\langle \text{STATEMENT} \rangle \rightarrow \text{id} \langle \text{ID_STATEMENT_CONTINUATION} \rangle \mid \langle \text{CONDITION} \rangle$
 $\mid \langle \text{CYCLE} \rangle \mid \langle \text{Print} \rangle$

$\langle \text{ID_STATEMENT_CONTINUATION} \rangle \Rightarrow = \langle \text{EXPRESSION} \rangle ;$

$\mid (\langle \text{EXPRESSION_OPCIONAL} \rangle) ;$

$\langle \text{EXPRESSION_OPCIONAL} \rangle \rightarrow \langle \text{EXPRESSION} \rangle \langle \text{MAS_EXPRESIONES} \rangle \mid \epsilon$

$\langle \text{MAS_EXPRESIONES} \rangle \rightarrow , \langle \text{EXPRESSION} \rangle \langle \text{MAS_EXPRESIONES} \rangle \mid \epsilon$

$\langle \text{ASSIGN} \rangle$

$\langle \text{ASSIGN} \rangle \rightarrow \text{id} = \langle \text{EXPRESSION} \rangle ;$

$\langle \text{CONDITION} \rangle$

$\langle \text{CONDITION} \rangle \rightarrow \text{if} (\langle \text{EXPRESSION} \rangle) \langle \text{Body} \rangle$
 $\langle \text{OPTIONAL_ELSE} \rangle ;$

$\langle \text{OPTIONAL_ELSE} \rangle \rightarrow \text{else} \langle \text{Body} \rangle \mid \epsilon$

$\langle \text{CYCLE} \rangle$

$\langle \text{CYCLE} \rangle \rightarrow \text{while } (\langle \text{EXPRESION} \rangle) \text{ do } \langle \text{Body} \rangle ;$

$\langle \text{F_Call} \rangle$

$\langle \text{F_Call} \rangle \rightarrow \text{id } (\langle \text{EXPRESION_OPCIONAL} \rangle) ;$

$\langle \text{EXPRESION_OPCIONAL} \rangle$ ya fue declarada en $\langle \text{STATEMENT} \rangle$

$\langle \text{Print} \rangle$

$\langle \text{Print} \rangle \rightarrow \text{print } (\langle \text{PRINT_LIST} \rangle) ;$

$\langle \text{PRINT_LIST} \rangle \rightarrow \langle \text{PRINT_ITEM} \rangle \langle \text{PRINT_MORE_ITEMS} \rangle$

$\langle \text{PRINT_ITEM} \rangle \rightarrow \langle \text{EXPRESION} \rangle \mid \text{cte.string}$

$\langle \text{PRINT_MORE_ITEMS} \rangle \rightarrow / \langle \text{PRINT_ITEM} \rangle \langle \text{PRINT_MORE_ITEMS} \rangle \mid \epsilon$

$\langle \text{EXPRESSION} \rangle$

$\langle \text{EXPRESSION} \rangle \rightarrow \langle \text{EXP} \rangle \langle \text{OPTIONAL_COMPARISON} \rangle$

$\langle \text{OPTIONAL_COMPARISON} \rangle \rightarrow \langle \text{RELATIONAL_OPERATOR} \rangle \langle \text{EXP} \rangle \mid \epsilon$

$\langle \text{RELATIONAL_OPERATOR} \rangle \rightarrow > \mid < \mid !=$

$\langle \text{EXP} \rangle$

$\langle \text{EXP} \rangle \rightarrow \langle \text{TERMINO} \rangle \langle \text{EXP}' \rangle$

$\langle \text{EXP}' \rangle \rightarrow + \langle \text{TERMINO} \rangle \langle \text{EXP}' \rangle \mid - \langle \text{TERMINO} \rangle \langle \text{EXP}' \rangle \mid \epsilon$

$\langle \text{TERMINO} \rangle$

$\langle \text{TERMINO} \rangle \rightarrow \langle \text{FACTOR} \rangle \langle \text{TERMINO}' \rangle$

$\langle \text{TERMINO}' \rangle \rightarrow * \langle \text{FACTOR} \rangle \langle \text{TERMINO}' \rangle$

$\mid / \langle \text{FACTOR} \rangle \langle \text{TERMINO}' \rangle \mid \epsilon$

<FACTOR>

<FACTOR> → (<EXPRESSION>) | <SIGN_EXPRESSION>

<SIGN_EXPRESSION> → <SIGN><DATA>

<SIGN> → + | - | %

<DATA> → id | <CTE>

<CTE>

<CTE> → cte_int | cte_float

BabyDuck

Entrega 0

Gramáticas del PDF 2

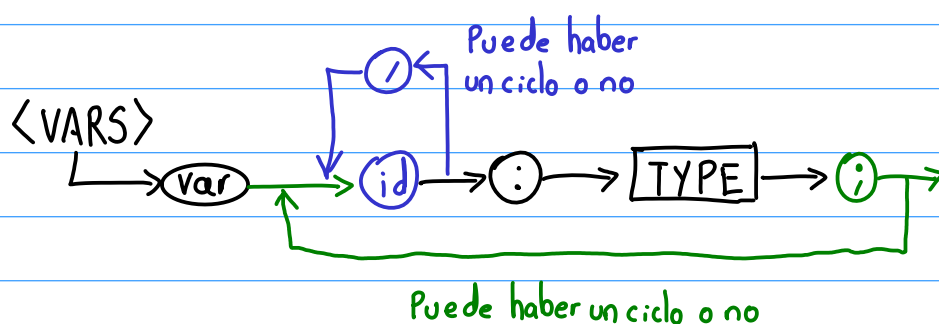
La definición de todas las siguientes gramáticas se realizó con el segundo PDF que compartió la profesora, el de Agosto-Diciembre 2025. Tiene variaciones que producen un lenguaje diferente, además de que las reglas y tokens están escritas en Español, no en Inglés como en el anterior

$\langle \text{Programa} \rangle$

$\langle \text{Programa} \rangle \rightarrow \text{programa id ; } \langle \text{VARIABLES_OPCIONALES} \rangle$
 $\langle \text{FUNCIONES_OPCIONALES} \rangle \text{ inicio } \langle \text{CUERPO} \rangle \text{ fin}$

$\langle \text{VARIABLES_OPCIONALES} \rangle \rightarrow \langle \text{VARs} \rangle \mid \epsilon$

$\langle \text{FUNCIONES_OPCIONALES} \rangle \rightarrow \langle \text{FUNCS} \rangle \langle \text{FUNCIONES_OPCIONALES} \rangle \mid \epsilon$



$\langle \text{VARs} \rangle$

$\langle \text{VARs} \rangle \rightarrow \text{vars } \langle \text{DECLARAR_VARIABLES} \rangle$

$\langle \text{DECLARAR_VARIABLES} \rangle \rightarrow \langle \text{DECLARAR_IDS} \rangle : \langle \text{TIPO} \rangle ;$
 $\langle \text{DECLARAR_VARIABLES}' \rangle$

$\langle \text{DECLARAR_VARIABLES}' \rangle \rightarrow \langle \text{DECLARAR_VARIABLES} \rangle \mid \epsilon$

$\langle \text{DECLARAR_IDS} \rangle \rightarrow \text{id } \langle \text{DECLARAR_IDS}' \rangle$

$\langle \text{DECLARAR_IDS}' \rangle \rightarrow , \langle \text{DECLARAR_IDS} \rangle \mid \epsilon$

$\langle \text{FUNCS} \rangle$

$\langle \text{FUNCS} \rangle \rightarrow \langle \text{TIPO_DE_RETORNO} \rangle \text{ id } (\langle \text{PARAMETROS} \rangle)$
 $\{ \langle \text{VARIABLES_LOCALES_OPCIONALES} \rangle \langle \text{CUERPO} \rangle \} ;$

$\langle \text{TIPO_DE_RETORNO} \rangle \rightarrow \text{nula} \mid \langle \text{TIPO} \rangle$

$\langle \text{PARAMETROS} \rangle \rightarrow \langle \text{LISTA_DE_PARAMETROS} \rangle \mid \epsilon$

$\langle \text{LISTA_DE_PARAMETROS} \rangle \rightarrow \text{id} : \langle \text{TIPO} \rangle \langle \text{MAS_PARAMETROS} \rangle$

$\langle \text{MAS_PARAMETROS} \rangle \rightarrow , \langle \text{LISTA_DE_PARAMETROS} \rangle \mid \epsilon$

$\langle \text{VARIABLES_LOCALES_OPCIONALES} \rangle \rightarrow \langle \text{VARs} \rangle \mid \epsilon$

<CUERPO>

<CUERPO> \rightarrow { <LISTA_DE_ESTATUTO> }

<LISTA_DE_ESTATUTO> \rightarrow <ESTATUTO> <LISTA_DE_ESTATUTO> | ϵ

<TIPO>

<TIPO> \rightarrow 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> \rightarrow id <CONTINUACION_DE_ESTATUTO_ID> | <CONDICION>
| <CICLO> | <IMPRIME>
| [<ESTATUTO OPCIONAL>]

<CONTINUACION_DE_ESTATUTO_ID> \rightarrow = <EXPRESION> ;
| (<EXPRESION OPCIONAL>) ;

<EXPRESION OPCIONAL> \rightarrow <EXPRESION> <MAS_EXPRESIONES> | ϵ

<MAS_EXPRESIONES> \rightarrow , <EXPRESION> <MAS_EXPRESIONES> | ϵ

<ESTATUTO OPCIONAL> \rightarrow <ESTATUTO> <ESTATUTO OPCIONAL> | ϵ

<ASIGNA>

<ASIGNA> \rightarrow id = <EXPRESION> ;

<CONDICION>

<CONDICION> \rightarrow si (<EXPRESION>) <CUERPO>
<SINO OPCIONAL> ;

<SINO OPCIONAL> \rightarrow sino <CUERPO> | ϵ

<CICLO>

<CICLO> \rightarrow mientras (<EXPRESION>) haz <CUERPO> ;

<LLAMADA>

<LLAMADA> \rightarrow id (<EXPRESION_OPCIONAL>)

<EXPRESION_OPCIONAL> ya fue declarada en <ESTATUTO>

<IMPRIME>

<IMPRIME> \rightarrow escribe (<IMPRIMIR_LISTA>) ;

<IMPRIMIR_LISTA> \rightarrow <IMPRIMIR_ELEMENTO> <IMPRIMIR_MAS_ELEMENTOS>

<IMPRIMIR_ELEMENTO> \rightarrow <EXPRESION> | letrero

<IMPRIMIR_MAS_ELEMENTOS> \rightarrow , <IMPRIMIR_ELEMENTO>

<IMPRIMIR_MAS_ELEMENTOS> | ϵ

<EXPRESION>

<EXPRESION> \rightarrow <EXP> <COMPARACION_OPCIONAL>

<COMPARACION_OPCIONAL> \rightarrow <OPERADOR_RELACIONAL> <EXP> | ϵ

<OPERADOR_RELACIONAL> \rightarrow > | < | != | ==

<EXP>

<EXP> \rightarrow <TERMINO> <EXP'>

<EXP'> \rightarrow + <TERMINO> <EXP'> | - <TERMINO> <EXP'> | ϵ

<TERMINO>

<TERMINO> \rightarrow <FACTOR> <TERMINO'>

<TERMINO'> \rightarrow * <FACTOR> <TERMINO'>

| / <FACTOR> <TERMINO'> | ϵ

<FACTOR>

<FACTOR> \rightarrow (<EXPRESSION>) | <SIN_EXPRESSION>

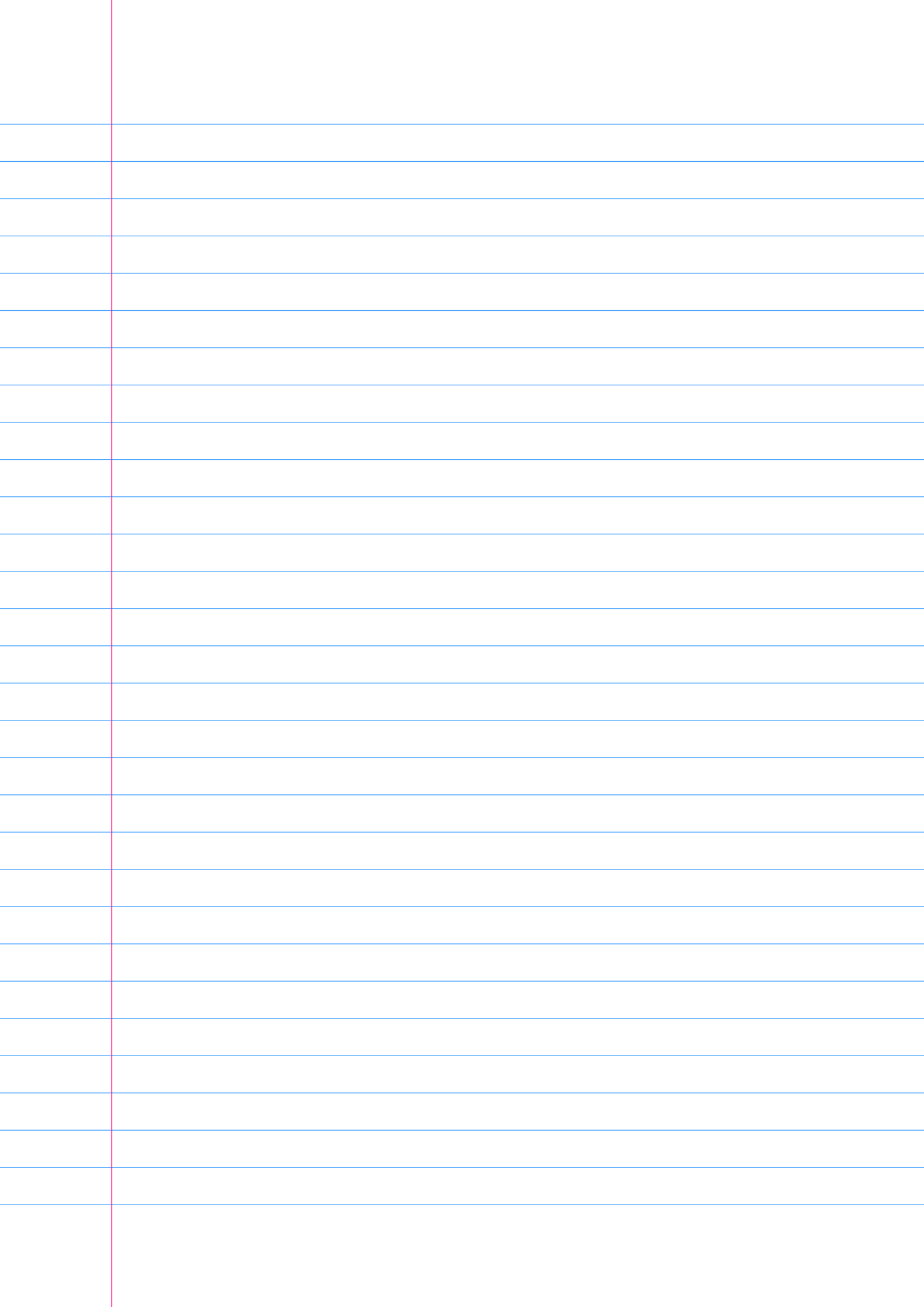
<SIN_EXPRESSION> \rightarrow <SIGNO> <DATO> | <LLAMADA>

<SIGNO> \rightarrow + | - | %

<DATO> \rightarrow id | <CTE>

<CTE>

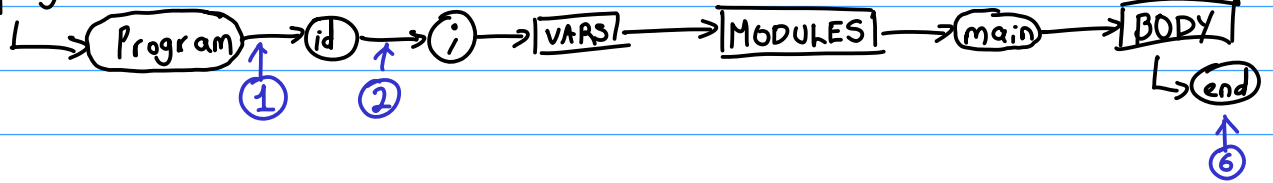
<CTE> \rightarrow cte_ent | cte_flot



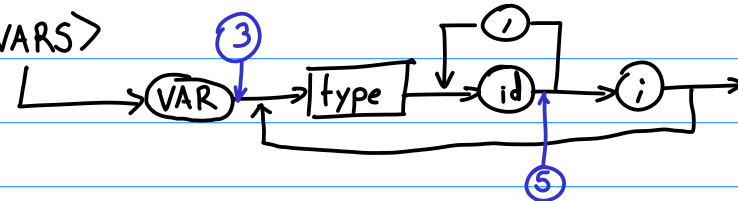
2025/10/16

Insert custom code

<prog>

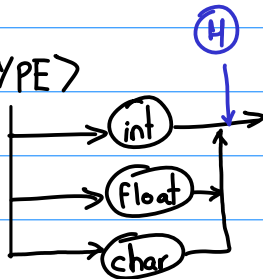


<VARS>



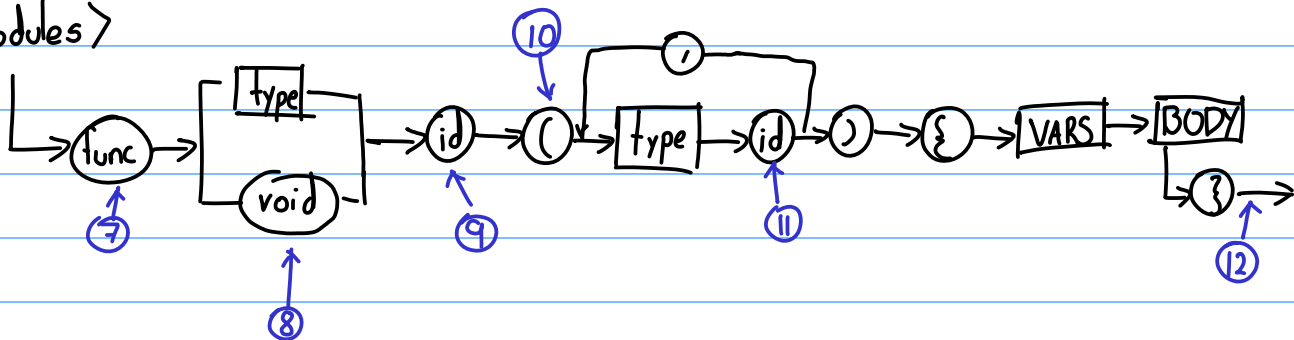
Neuralgic action: stop execution here and define custom rules

<TYPE>



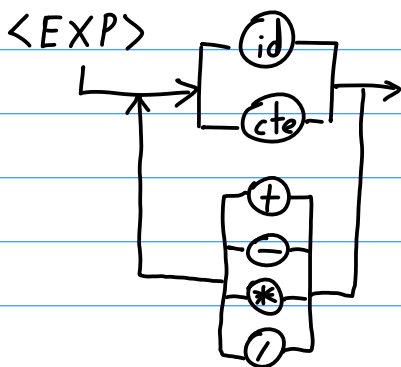
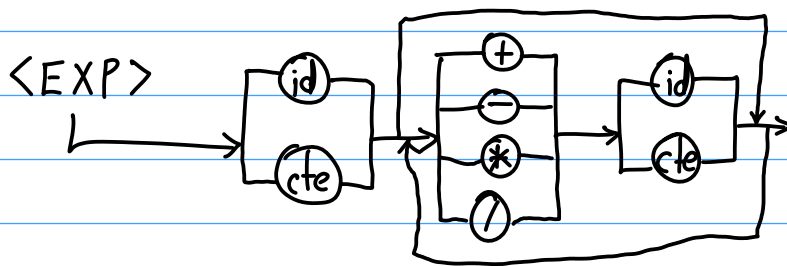
1. Create DirFunc
2. Add id-name and type program a DirFunc
3. If current Func doesn't have a VarTable then create VarTable and link it to current Func
4. Current-type = type
5. Search for id-name in current VarTable
if found → Error "multiple declaration"
if not, add id-name and current-type to current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

<Modules>



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. Create 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

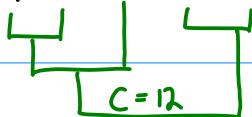


- Both produce $A+B$, A , $A+B * C$
- Both expressions are equivalent but not the same
- The syntax only cares that we have the exact sequence

La asociatividad entra en juego cuando no hay jerarquía

Asociativo Izquierdo

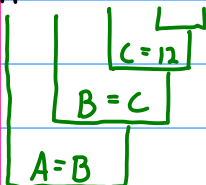
$$A = B = C = 3 * 4$$



para que esta operación funcione se debe usar la asociatividad derecha

Asociativo Derecho

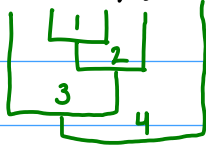
$$A = B = C = 3 * 4$$



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

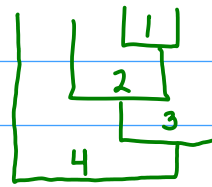
Left Associative

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

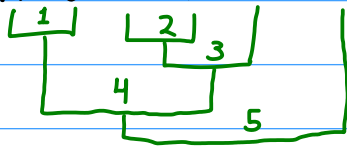


Right Associative

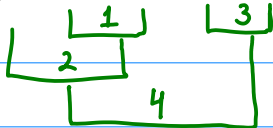
$$A + B * C / D - E$$



$$A * B + C / D * F + G$$



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



CÓDIGO INTERMEDIO

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

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

$A + B \rightarrow AB +$

$A + B * C \rightarrow ABC * +$

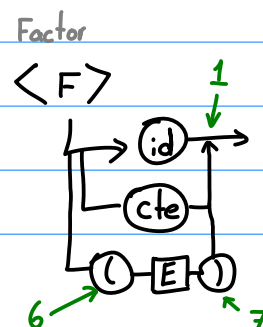
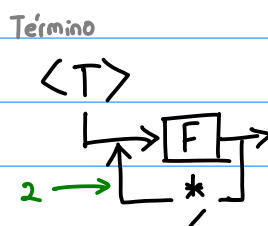
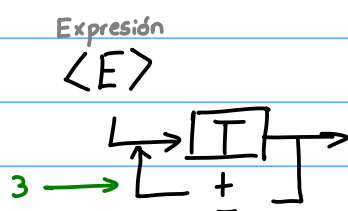
$A * B * C \rightarrow$

Asociativa Izquierda $\rightarrow AB * C *$

Asociativa Derecha $\rightarrow \cancel{ABC} * *$

$A + B * C / D * E$
 $\hookrightarrow ABC * D / E * +$

Todos los ejercicios siguen las siguientes reglas:



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

\Rightarrow

$op = poper.pop()$

$vp.push(op)$

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

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

$A + B * C \rightarrow *, B, C, t_1$
 $+ , A, t_1, t_2$

$A + B * C / D \rightarrow *, B, C, t_1$
 $/, t_1, D, t_2$
 $+ , A, t_2, t_3$

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

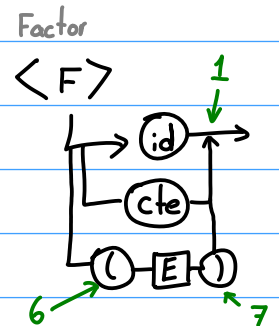
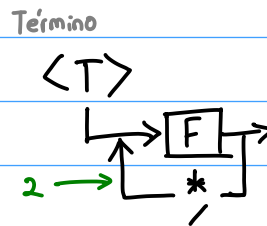
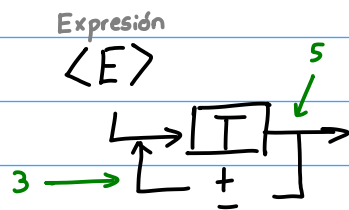
popers
~~[+, *, /]~~

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:



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

\Rightarrow

$op = poper.pop()$

$der = Pvar.pop()$

$izq = Pvar.pop()$

$t_1 = pedir temporal$

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

int int float int int float float int int float

$poper$

$[* (+ / * (+ *]$

$Pvars$

$[A \ B \ C \ D \ t_1 \ F \ G \ I \ t_2 \ t_3 \ t_4]$

int int float int float int float float float float float 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

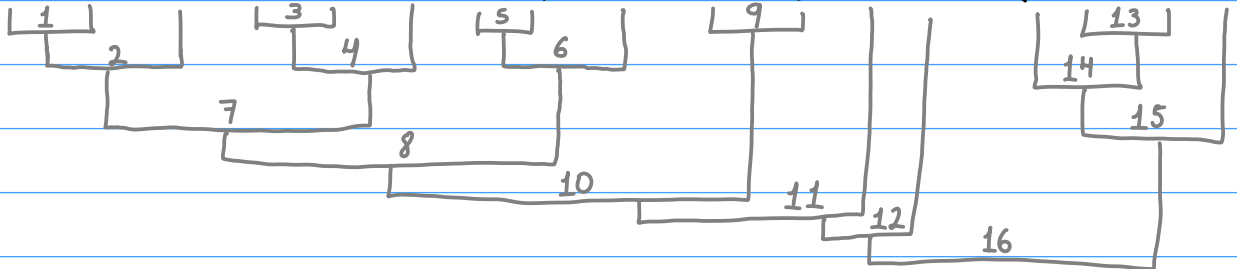
2025/11/04

Tarea 3. Expresiones

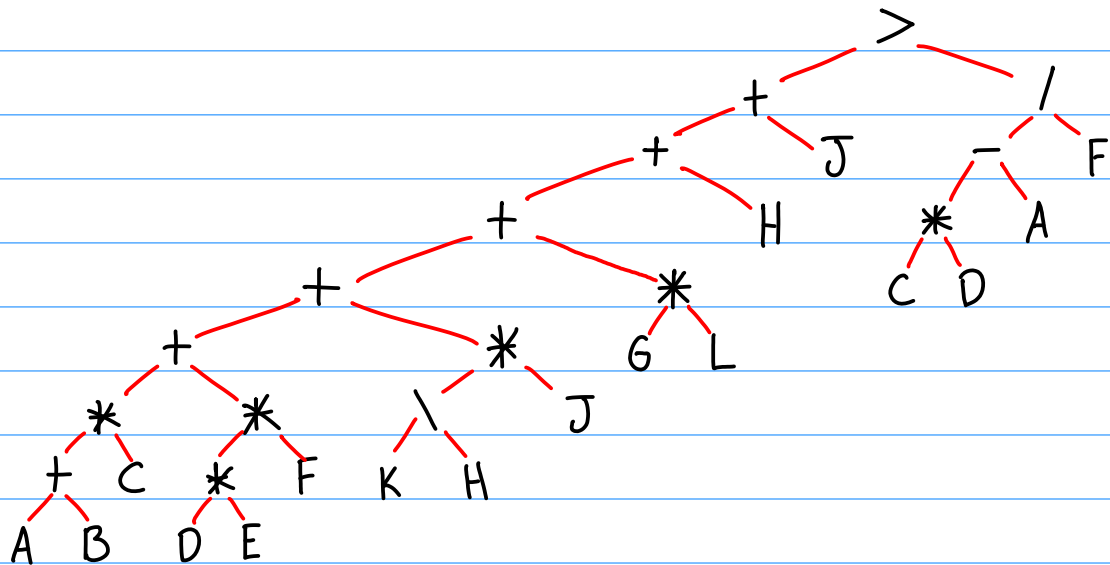
$$((A+B)*C + D*E*F + K/H * J) + G * L + H + J > (A-C*D)/F$$

Árbol de evaluación asociativo izquierdo

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

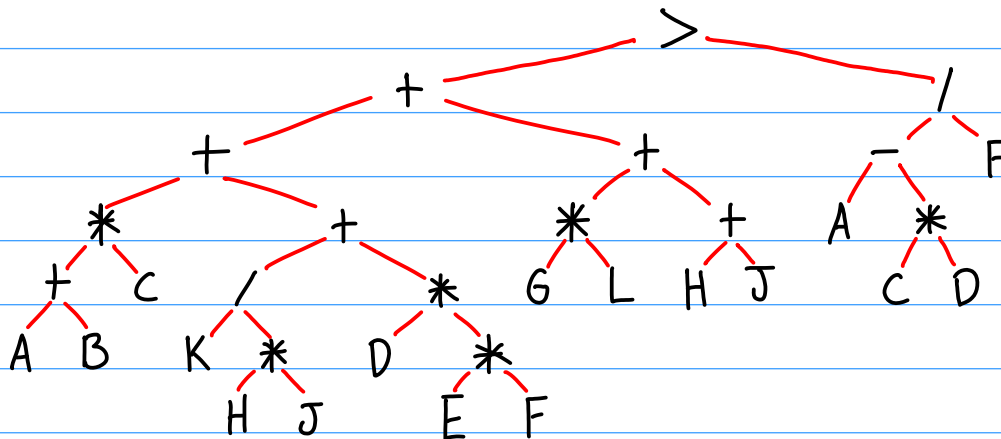
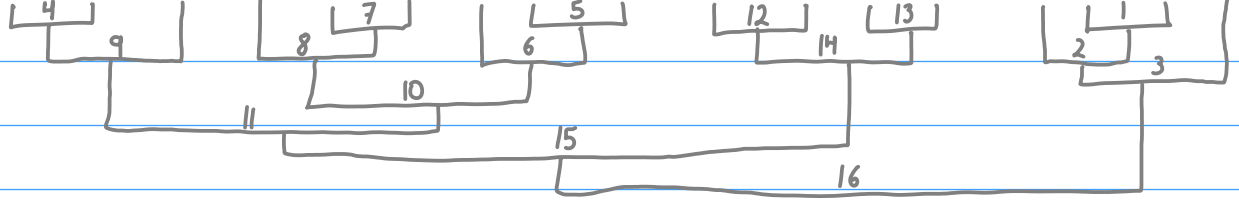


El árbol se construye con un recorrido post-order, la raíz y los nodos cercanos a ella son los operadores con menor precedencia mientras que las hojas y los nodos cercanos a ellas contienen los operadores de mayor precedencia



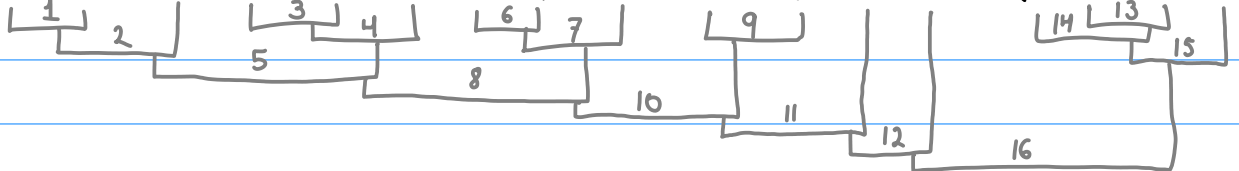
Árbol de evaluación asociativo derecho

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



Vector Polaco asociativo izquierdo

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



popper

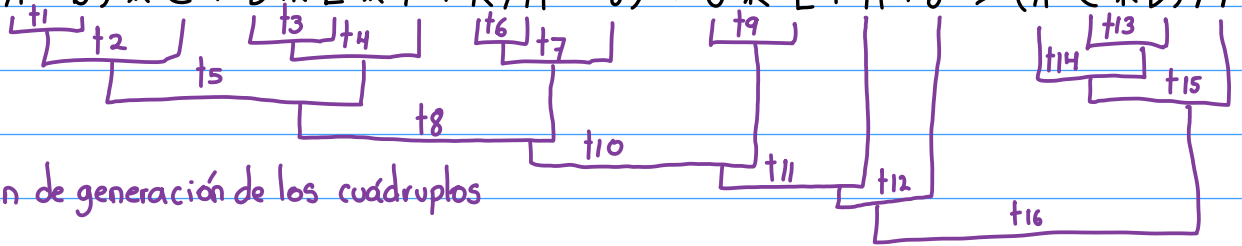
$$[(\frac{1}{1}(\frac{2}{2}(\frac{3}{3}(\frac{4}{4}(\frac{5}{5}(\frac{6}{6}(\frac{7}{7}(\frac{8}{8}(\frac{9}{9}(\frac{10}{10}(\frac{11}{11}(\frac{12}{12}(\frac{13}{13}(\frac{14}{14}(\frac{15}{15}(\frac{16}{16})))))))))))]$$

vp

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

Cuádruplos asociativo izquierdo

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



Orden de generación de los cuádruplos

paper

$$[(\cancel{(\cancel{+})} \times \cancel{+} \times \cancel{*} \cancel{+} \cancel{*} \cancel{*} \cancel{+} \cancel{/} \cancel{*} \cancel{)} \cancel{+} \cancel{*} \cancel{+} \cancel{+} \cancel{+} \cancel{+} \cancel{+} \cancel{-} \cancel{*} \cancel{)} \cancel{/}]$$

$$\quad \quad \quad t_1 \quad t_2 \quad t_5 \quad t_3 \quad t_4 \quad t_8 \quad t_6 \quad t_7 \quad t_{10} \quad t_9 \quad t_{11} \quad t_{12} \quad t_{16} \quad t_{14} \quad t_{13} \quad t_{15}$$

Prar's

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

Cuádruplos

$$+ A \quad B \quad + 1$$
$$* t_1 \subset t_2$$

* D E t₃

$$* \quad t_3 \quad F \quad t_4$$
$$t \quad t_2 \quad t_4 \quad t_5$$
$$/ K H t_6$$
$$* t_6 \quad J \quad t_7$$
$$t \quad t_5 \quad t_7 \quad t_8$$

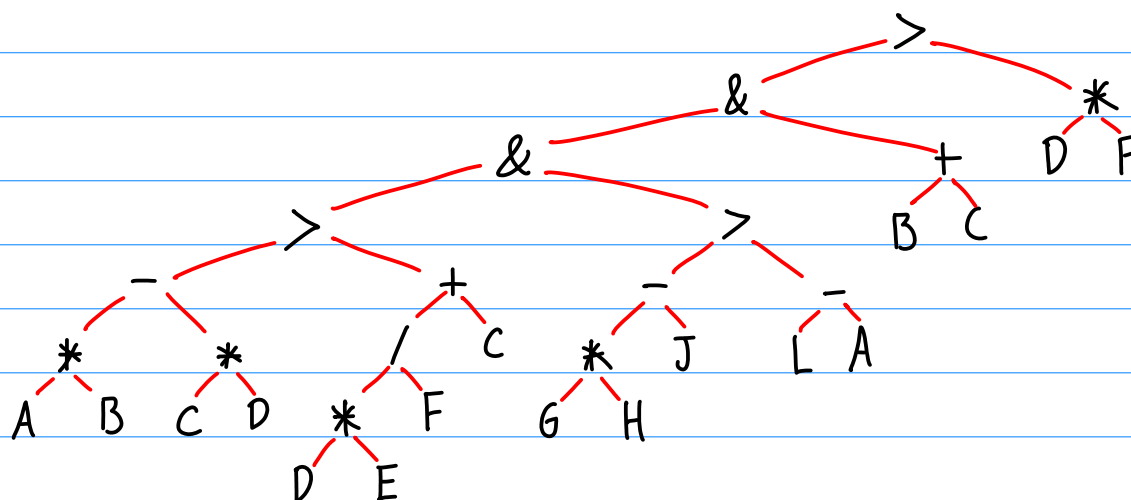
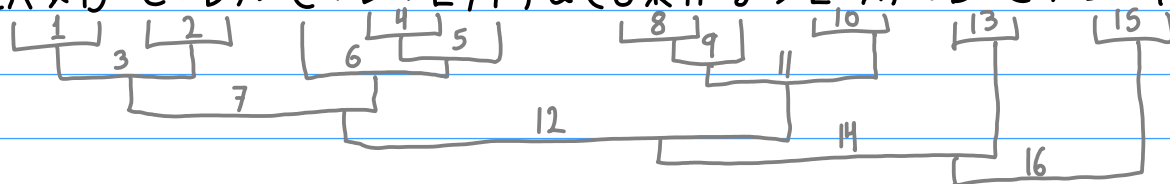
* 6 L t₉

$$t \quad t_8 \quad t_9 \quad t_{10}$$
$$+ t_{10} \quad H \quad t_{11}$$
$$+ t_{11} \quad J \quad t_{12}$$
$$* \quad C \quad D \quad t_{13}$$
$$-A \quad t_{13} \quad t_{14}$$
$$/ \quad t_{14} \quad F \quad t_{15}$$
$$> t_{12} \quad t_{15} \quad t_{16}$$

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

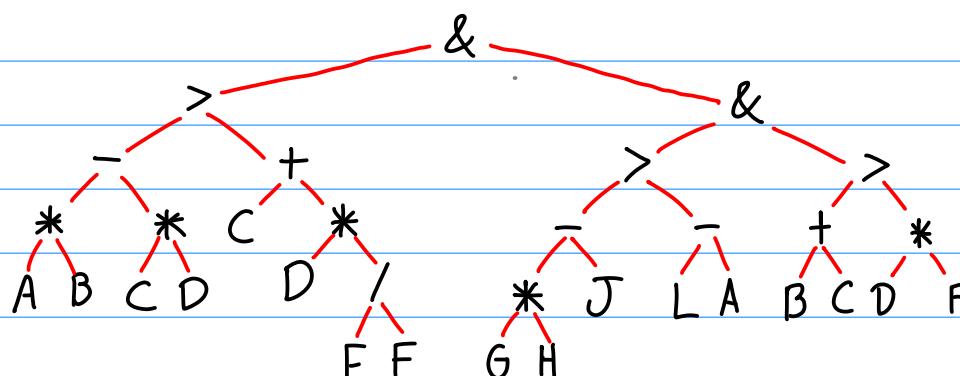
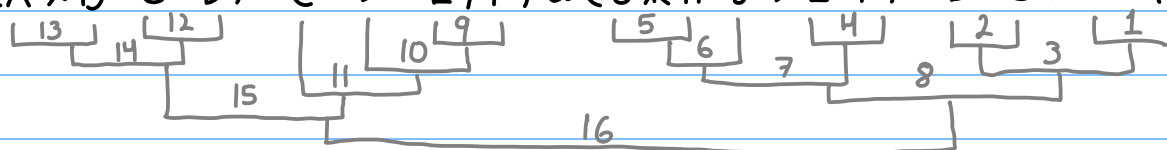
Árbol de evaluación asociativo izquierdo

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



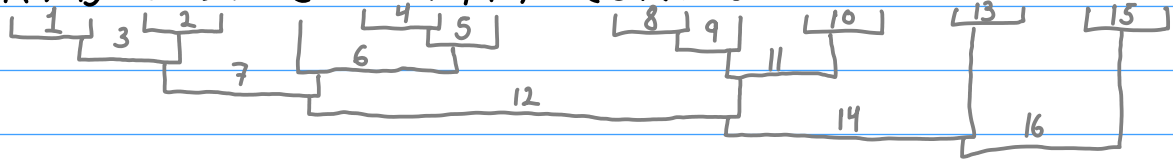
Árbol de evaluación asociativo derecho

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



Vector Polaco asociativo izquierdo

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



popper

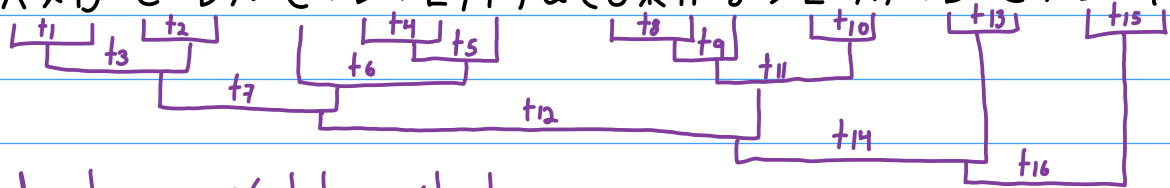
$$[(\cancel{A} * \cancel{B} - \cancel{C} * \cancel{D}) > \cancel{C} + \cancel{D} * \cancel{E} / \cancel{F}] \& (\cancel{G} * \cancel{H} - \cancel{J} > \cancel{L} - \cancel{A}) \& \cancel{B} + \cancel{C} > \cancel{D} * \cancel{F}]$$

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

paper

$$[(\ (\ * \ - \ * \) \ > \ + \ * \ / \) \ \& \ (\ * \ - \ > \ + \) \ \& \ + \ > \ * \]$$

t_1 t_3 t_2 t_7 t_6 t_4 t_5 t_{12} t_8 t_9 t_{11} t_{10} t_{14} t_{13} t_{16} t_{15}

Prvar's

$$[A \ B \ t_1 \ C \ D \ t_2 \ t_3 \ C \ D \ E \ t_4 \ F \ t_5 \ t_6 \ t_7 \ G \ H \ t_8 \ J \ t_9 \ L \ A \ t_{10} \ t_{11} \ t_{12} \ B \ C \ t_{13} \ t_{14} \ D \ F \ t_{15} \ t_{16}]$$

Cuádruplos

$$* \quad A \quad B \quad t_1$$

$$* \quad C \quad D \quad t_2$$

$$- \quad t_1 \quad t_2 \quad t_3$$

$$* \quad D \quad E \quad t_4$$

$$/ \quad t_4 \quad F \quad t_5$$

$$+ \quad C \quad t_5 \quad t_6$$

$$> \quad t_3 \quad t_6 \quad t_7$$

$$* \quad G \quad H \quad t_8$$

$$- \quad t_8 \quad J \quad t_9$$

$$- \quad L \quad A \quad t_{10}$$

$$> \quad t_9 \quad t_{10} \quad t_{11}$$

$$\& \quad t_7 \quad t_{11} \quad t_{12}$$

$$+ \quad B \quad C \quad t_{13}$$

$$\& \quad t_{12} \quad t_{13} \quad t_{14}$$

$$* \quad D \quad F \quad t_{15}$$

$$> \quad t_{14} \quad t_{15} \quad t_{16}$$

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

GOTO

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 t₁

2. * C D t₂

3. > t₁ t₂ t₃

4. gotoF t₃ 8

5. + B C t₄

6. = t₄ A

7. goto 11

8. - D A t₅

9. = t₅ C

10. print C

11. + D E t₆

12. = t₆ B

13.

Si t₃ es False moverse al cuádruplo 8, en caso contrario continuar desde el cuádruplo 5 en adelante

	if ₁	if ₂	while ₁	else ₂	else ₁	while ₂	
Stack de saltos: [5 , 8 , 9 , 11 , 18 , 21 , 22 , 23]							
if (A * B - C > D + E) {	1. *	A	B	t ₁	21. goto	28	fin del if 1 salir
if (B > C + D) {	2. -	t ₁	C	t ₂	22. >	A	B t ₁₂
while (A + B > D) {	3. +	D	E	t ₃	23. gotoF	t ₁₂	28
A = A - C;	4. >	t ₂	t ₃	t ₄	24. -	B	C t ₁₃
print(A, B + C);	5. gotoF	t ₄		22	25. =	t ₁₃	B
}	6. +	C	D	t ₅	26. print		C
}	7. >	B	t ₅	t ₆	27. goto	22	
else { B = C + D; }	8. gotoF	t ₆		19	28. *	C	D t ₁₄
}	9. +	A	B	t ₇	29. =	t ₁₄	B
else {	10. >	t ₇	D	t ₈	30. *	C	D t ₁₅
while (A > B) {	11. gotoF	t ₈		18	31. +	B	t ₁₅ t ₁₆
B = B - C;	12. -	A	C	t ₉	32. =	t ₁₆	A
print(C);	13. =	t ₉		A	33.		
}	14. print			A	34.		
}	15. +	B	C	t ₁₀	35.		
B = C * D;	16. print			t ₁₀			
A = B + C * D;	17. goto			9			
	18. goto	21					
	19. +	C	D	t ₁₁			
	20. =	t ₁₁		B			

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 $A * B - C \geq D * E / (G + H) \{$

$H = J * K + B;$

 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 = $[15, 20, 25, 34, 35]$

	while 1	else 1	while 2	fin del if 1	do
1.	/		E	F	t ₁
2.	-		D		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
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.	GOTO		t ₁₂		45
16.	*		J	K	t ₁₃
17.	+		t ₁₃	B	t ₁₄
18.	=		t ₁₄		H
19.	<		B	H	t ₁₅
20.	GOTO		t ₁₅		35
21.	+		H	J	t ₁₆
22.	=		t ₁₆		B
23.	+		A	C	t ₁₇
24.	>		B	t ₁₇	t ₁₈
25.	GOTO		t ₁₈		34
26.	*		B	C	t ₁₉

Evaluar WHILE 1

Terminar WHILE 1
Continuar WHILE 1

IF 1
expresión

ELSE 1

IF 1
EXP = True

Evaluar WHILE 2

Terminar WHILE 2
Continuar WHILE 2

27. + A t₁₉ t₂₀

28. PRINT t₂₀

29. - D E t₂₁

30. PRINT t₂₁

31. - B J t₂₂

32. = t₂₂ B

Re-evaluar
WHILE 2

33. GOTO 23

Fin del
if 1

34. GOTO 44

Primer cuádruplo
del DO-WHILE

35. + A B t₂₃

36. = t₂₃ A

37. - B D t₂₄

38. PRINT t₂₄

Evaluar WHILE
de DO-WHILE

39. - A D t₂₅

40. + C B t₂₆

41. < t₂₅ t₂₆ t₂₇

Repetir lógica
del DO

42. GOTOV t₂₇ 35

Salir del
DO-WHILE

43. GOTO 44

Fin del IF-ELSE 1

Re-evaluar WHILE 1

44. GOTO 9

45. + A B t₂₈

46. = t₂₈ F

2025/11/22

Tarea 5: Módulos Generación de Código Intermedio

GOTO
main()1er else
uno()fin del 1er
if → uno()while Falso
dos()DO
main()

```
program patito;
var
```

stack de saltos = [~~1~~, ~~3~~, ~~11~~, ~~17~~, ~~36~~]

int i, j, K;	1.	GOTO		32	
float f;	2.	>	a	0	t1
	3.	GOTO		t1	12
void uno(int a, int b) {	4.	*	b	j	t2
if (a > 0) {	5.	+	a	t2	t3
i = a + b * j + i;	7.	+	t3	i	t4
print(i + j);	8.	=	t4		i
}	9.	+	i	j	t5
else {	10.	PRINT			t5
print(a + b);	11.	GOTO		14	
}	12.	+	a	b	t6
}	13.	PRINT			t6
	14.	ENDFUNC			
void dos(int a, float g) {	15.	=	a		i
int i;	16.	>	a	0	t7
i = a;	17.	GOTO		t7	31
while (a > 0) {	18.	*	K	j	t8
a = a - K * j;	19.	-	a	t8	t9
uno(a * 2, a + K);	20.	=	t9		a
g = g * j - K;	21.	ERA		uno	
}	22.	*	a	2	t10
}	23.	PARAM		t10	PARAM1
	24.	+	a	K	t11
main() {	25.	PARAM		t11	PARAM2
i = 2, K = i + 1;	26.	GOSUB		uno	
f = 3.14;	27.	*	g	j	t12

Ejecutar else 1

Ejecutar if 1

Fin del if 1

Fin del 1er
if-elsefin de
uno()1er cuádruplo del
while de dos()

	do {	28.	-	t12	K	t13
	dos(i+k, f*3);	29.	=	t13		g
	print(i, j*2, f*2+1.5);	30.	GOTO		16	
	i=i-k/2;	31.	ENDFUNC			
	}	32.	=	2		i
→	while(i>0);	33.	+	i	1	t14
	}	34.	=	t14		K
		35.	=	3.14		f
		36.	ERA		dos	
		37.	+	i	K	t15
		38.	PARAM		t15	PARAM1
		39.	*	f	3	t16
		40.	PARAM		t16	PARAM2
		41.	GOSUB		dos	
		42.	PRINT			i
		43.	*	j	2	t17
		44.	PRINT			t17
		45.	*	f	2	t18
		46.	+	t18	1.5	t19
		47.	PRINT			t19
		48.	/	K	2	t20
	while(i>0);	49.	-	i	t20	t21
		50.	=	t21		i
		51.	>	i	0	t22
		52.	GOTOV		t22	36
		53.	GOTO		54	
		54.	END			
		55.				
		56.				
		57.				

Re-evaluar while
de dos()

fin del while
de dos()

Primer cuádruplo del
DO-WHILE de main()

while(i>0);

Evaluar lógica del
WHILE del DO-WHILE
Repetir lógica
del DO
Salir del
DO-WHILE