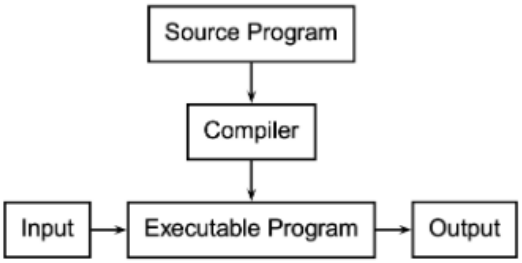
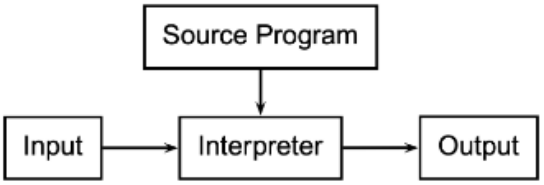
Introducció a la compilació

1. Processadors de llenguatges

Un **compilador** és un programa que tradueix programes escrits en un LP d'alt nivell a codi màquina (o, en general, a codi de baix nivell). Exemples: GCC, CLANG, go, ghc, …



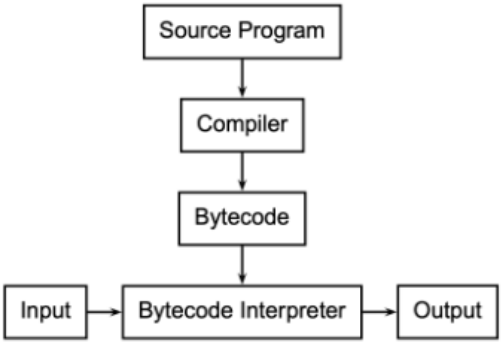
Un **intèrpret** és un programa que executa directament instruccions escrites en un LP. Exemples: Python, PHP, Perl, ghci, BASIC, Logo, …



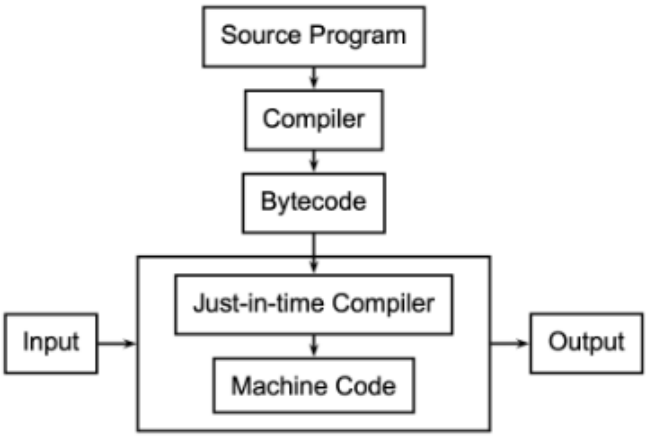
Intèrprets de bytecode: variant entre els compiladors i els intèrprets:

- El bytecode és un codi intermig més abstracte que el codi màquina.
- Augmenta la portabilitat i seguretat i facilita la interpretació.
- Una màquina virtual interpreta programes en bytecode.

Exemples: Java, Python, …



Compiladors just-in-time: la compilació just-in-time (JIT) compila fragments del programa durant la seva execució. Un analitzador inspecciona el codi executat per veure quan val la pena compilar-lo. Exemples: Julia, V8 per Javascript, JVM per Java, …

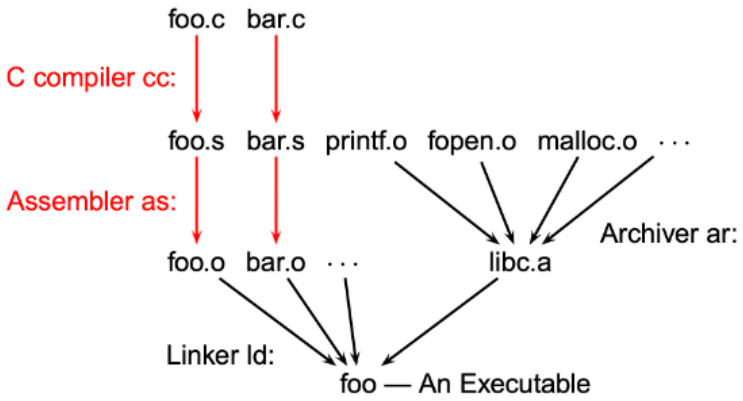


Preprocessadors: prepara el codi font d'un programa abans que el compilador el vegi:

- Expansió de macros.
- Inclusió de fitxers.
- Compilació condicional.
- Extensions de llenguatge.

Exemples: cpp, m4, …

Ecosistema: els processadors de llenguatges viuen en un ecosistema gran i complex (preprocessadors, compiladors, enllaçadors, gestors de llibreries, ABIs, formats d'executables, …).



2. Sintaxi

La sintaxi d'un llenguatge de programació és el conjunt de regles que defineixen les combinacions de símbols que es consideren construccions correctament estructurades.

Sovint s'especifica la sintaxi utilitzant una gramàtica lliure de context (context-free grammar).

Els elements més bàsics s'especifiquen a través d'expressions regulars.

```
expr → NUM
     | '(' expr ')'
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr

NUM  → [0-9]+ ( '.' [0-9]+ )
```

3. Semàntica

La semàntica d'un LP descriu què significa un programa ben construït.

A vegades, les construccions sintàcticament correctes poden ser semànticament incorrectes.

Hi ha bàsicament dues maneres d'especificar formalment la semàntica:

- Semàntica operacional: defineix una màquina virtual i com l'execució del programa canvia l'estat de la màquina.
- Semàntica denotacional: mostra com construir una funció que representa el comportament del programa (és a dir, una transformació d'entrades a sortides) a partir de construccions del LP.

4. Flux de compilació

Etapes:

- Front end:
  - ➔ preprocessador.
  - ➔ analitzador lèxic (escàner).
  - ➔ analitzador sintàctic (parser).
  - ➔ analitzador semàntic.
- Middle end:
  - ➔ analitzador de codi intermig.
  - ➔ optimitzador de codi intermig.
- Back end:
  - ➔ generador de codi específic.
  - ➔ optimizador de codi específic.

L'analitzador lèxic (escàner) agrupa els caràcters en "paraules" (tokens) i elimina blancs i comentaris.

L'analitzador sintàctic (parser) construeix un arbre de sintaxi abstracta (AST) a partir de la seqüència de tokens i les regles sintàctiques. Les paraules clau, els separadors, parèntesis i blocs s'eliminen.

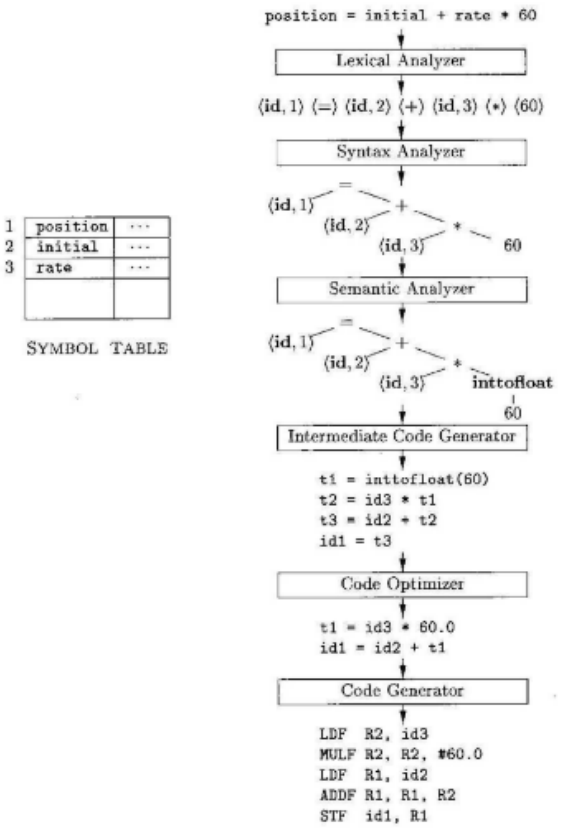
L'analitzador semàntic recorre l'AST i:

- crea la taula de símbols,
- assigna memòria a les variables,
- comprova errors de tipus,
- resol ambigüitats.

El resultat és la taula de símbols i un AST decorat.

El generador de codi tradueix el programa a codi de tres adreces (ensamblador idealitzat amb infinitat de registres).

El back end tradueix i optimitza el codi de tres adreces a l'arquitectura desitjada.



5. Eines

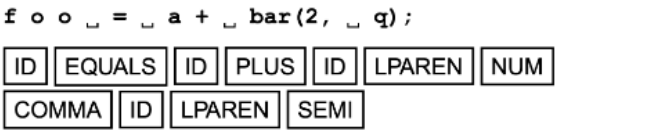
Per construir un compilador no es parteix de zero, hi ha moltes eines que donen suport:

- ANTLR, donades les especificacions lèxiques i sintàctiques del LP, construeix automàticament l'escàner, l'analitzador i l'AST.
- LLVM ofereix una col·lecció d'eines modulars reutilitzables pels backends dels compiladors.

Anàlisi lèxica

L'analitzador lèxic (o escàner) converteix una seqüència de caràcters en una seqüència de tokens:

- identificadors,
- literals (nombres, textos, caràcters),
- paraules clau,
- operadors,
- puntuació …



Token	Lexemes	Pattern
EQUALS	=	an equals sign
PLUS	+	a plus sign
ID	a foo bar	letter followed by letters or digits
NUM	0 42	one or more digits

1. Objectius

Els seus objectius son:

- Simplificar la feina de l'analitzador sintàctic.
- Descartar detalls irrellevants: blancs, comentaris, …
- Els escàners són molt més ràpids que els parsers.

2. Descripció de tokens

Per especificar els tokens s'utilitzen conceptes de la teoria de llenguatges:

- Alfabet: un conjunt finit de símbols.
- Paraula: una seqüència finita de símbols de l'alfabet.
- Llenguatge: un conjunt de paraules sobre un alfabet.

Operacions sobre llenguatges:

- Concatenació: una paraula d'un llenguatge seguida d'una paraula d'un altre llenguatge.
- Unió: totes les paraules de cada llenguatge.
- Clausura de Kleene: zero o més concatenacions.

3. Expressions regulars

Les **expressions regulars** descriuen llenguatges a partir de tokens sobre un alfabet Σ.

1. ε és una expressió regular que denota {ε}.
2. Si $a \in \Sigma$, a és una expressió regular que denota $\{a\}$.
3. Si r i s denoten els llenguatges $L(r)$ i $L(s)$,
   - $(r)|(s)$ denota $L(r) \cup L(s)$
   - $(r)(s)$ denota $\{tu : t \in L(r), u \in L(s)\}$
   - $(r\ ^*)$ denota la clausura transitiva de $L(r)$

Exemples:

$\Sigma = \{a, b\}$

RE	Language
$a\|b$	$\{a, b\}$
$(a\|b)(a\|b)$	$\{aa, ab, ba, bb\}$
a^*	$\{\epsilon, a, aa, aaa, aaaa, \ldots\}$
$(a\|b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \ldots\}$
$a\|a^*b$	$\{a, b, ab, aab, aaab, aaaab, \ldots\}$

4. Generadors d'escàners

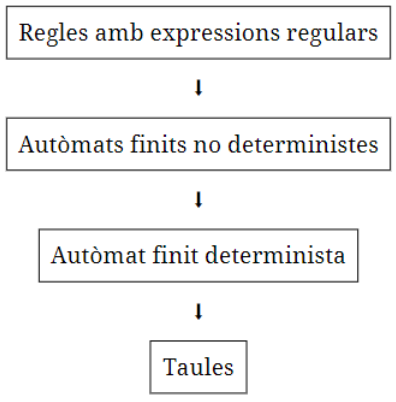
Les expressions regulars s'usen en eines per crear compiladors: lex, ANTLR, …

I també,

- en comandes del SO per tractar fitxers (grep, sed, …).
- en llibreries d'LPs per tractar textos (re en Python, …).

A partir de la definició lèxica, l'escàner és un autòmat determinista que produeix com a sortida els tokens reconeguts.

Construcció:



5. Analitzador lèxic d'ANTLR

```
RET     : 'return' ;

LPAREN  : '(' ;
RPAREN  : ')' ;

ADD     : '+' ;
SUB     : '-' ;
MUL     : '*' ;
DIV     : '/' ;

DIGIT   : '0'..'9' ;
LETTER  : [a-zA-Z] ;

NUMBER  : (DIGIT)+ ;
IDENT   : LETTER (LETTER | DIGIT)* ;

WS      : [ \t\n]+ -> skip ;
```

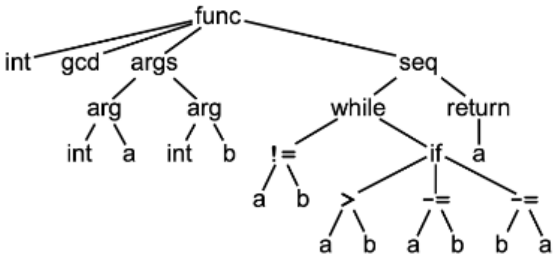
Referència

- els noms dels tokens han de ser en majúscules
- textos entre cometes representen aquell text
- ajuntar indica concatenació
- `|` indica unió
- `*` indica zero o més
- `+` indica un o més
- `?` indica un o cap
- `..` indica rangs
- es poden agrupar elements amb parèntesis
- l'`skip` no reporta el token

Anàlisi sintàctica

L'objectiu de l'analitzador sintàctic (o parser) és convertir una seqüència de tokens en un arbre de sintaxi abstracta que capturi la jerarquia de les construccions.

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```



- Es descarta informació no rellevant com paraules clau, els separadors, els parèntesis i els blocs.
- Es facilita la feina dels propers estadis.

1. Gramàtiques

La majoria dels LPs es descriuen a través de gramàtiques incontextuals, usant notació BNF (Backus-Naur form).

Les gramàtiques incontextuals permeten descriure llenguatges més amplis que els llenguatges regulars perquè són "recursives". La recursivitat permet definir jerarquies i niuar elements (parèntesis o blocs).

Però té una sèrie de dificultats:

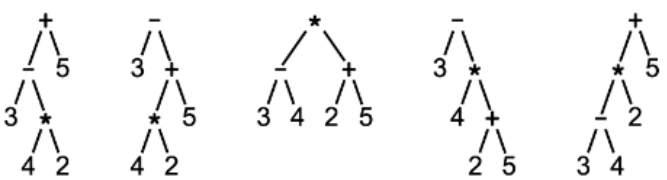
- Gramàtiques ambigües.
- Prioritat dels operadors.
- Associativitat dels operadors.
- Analitzadors top-down vs bottom-up.
- Recursivitat per la dreta / per l'esquerra.

2. Gramàtiques ambigües

Una gramàtica és ambigüa si un mateix text es pot derivar (organitzar en un arbre segons la gramàtica) de diferents maneres.

```
expr → expr + expr | expr - expr | expr * expr | NUM
```

el fragment `3 - 4 * 2 + 5` es pot derivar d'aquestes maneres:



Associar prioritat i associativitat als operadors sol permetre eliminar ambigüitats.

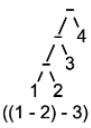
Prioritat: `1 * 2 + 3 * 4`

`*` té més prioritat que `+`

`+` té més prioritat que `*`

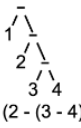
Associativitat: `1 - 2 - 3 - 4`

associativitat per l'esquerra



((1 - 2) - 3) - 4

associativitat per la dreta



1 - (2 - (3 - 4))

3. Generadors d'analitzadors sintàctics

Existeixen diferents tècniques per generar analitzadors sintàctics, amb propietats diferents:

- Analitzadors descendents (top-down parsers): reconeixen l'entrada d'esquerra a dreta tot buscant derivacions per l'esquerra expandint la gramàtica de l'arrel cap a les fulles.
- Analitzadors ascendents (bottom-up parsers): reconeixen primer sobrel les unitats més petites de l'entrada analitzada abans de reconèixer l'estructura sintàctica segons la gramàtica.

Analitzadors descendents LL(k):

- LL: Left-to-right, Left-most derivation
- k: nombre de tokens que mira endavant

Idea bàsica: mirar el següent token per poder decidir quina producció utilitzar.

Inconvenients principals:

- Les produccions no poden tenir prefixos comuns.
- Les regles no poden tenir recursivitat per l'esquerra.

ANTLR és un analitzador descendent LL(k).

```
expr  → expr '+' term
      | expr '-' term
      | term
```

⬇ s'escriu senzillament

```
expr : term ('+' term | '-' term) * ;
```

A més, la prioritat dels operadors ve donada per l'ordre d'escriptura:

```
expr : expr '*' expr
     | expr '+' expr
     | NUM ;
```

I es pot definir fàcilment l'associativitat:

```
expr : <assoc=right> expr '^' expr
     | NUM ;
```

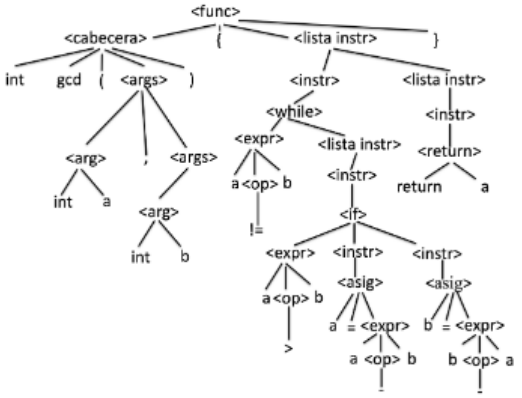
Arbres de sintaxi abstracta

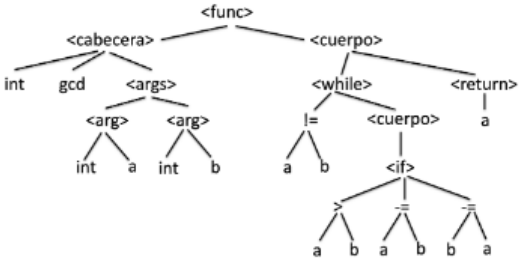
Amb un analitzador descendent, es poden executar accions durant el reconeixement de les regles, les accions poden aparèixer en qualsevol punt de la regla.

Amb un analitzador ascendent, només es poden executar accions després de reconèixer una regla.

Usualment, les accions construeixen un arbre de sintaxi concreta que segueix les regles de la gramàtica.

Aquest arbre es sol convertir en un arbre de sintàxi abstracta.





→ Es separa l'anàlisi de la traducció.
→ Es facilita les modificacions tot minimitzant les interaccions.
→ Es permet que diferents parts del programa s'analitzin en ordres diferents.

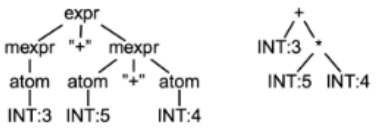
Arbre de sintaxi concreta / de derivació: reflecteix les regles sintàctiques.

Arbre de sintaxi abstracta: representa el programa fidelment, però eliminar i simplifica detalls sintàctics irrellevants.

Exemple: Eliminar regles per desambigüar gramàtica.

```
expr    : mexpr ('+' mexpr) * ;
mexpr   : atom  ('*' atom ) * ;
atom    : NUM ;
```

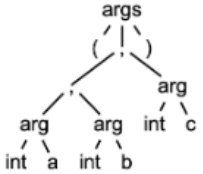
3 + 5 * 4

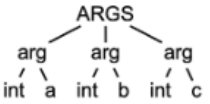


Concrete Parse Tree Abstract Syntax Tree

Exemple: Aplanar una llista de paràmetres.

```
int gcd(int a, int b, int c)
```





1. Ús dels ASTs

Un cop construït l'AST, les etapes següents el recorren per a dur a terme les seves tasques:

- L'anàlisi semàntica verificarà l'ús correcte dels elements del programa.
- El generador de codi visitarà l'arbre i li aplicarà regles per generar codi intermig.
- L'intèrpret es passejarà per l'arbre per dur a terme les seves instruccions.

2. ASTs en ANTLR

A partir de la gramàtica, ANTLR pot generar un analitzador descendent amb accions que construeixen un AST. L'AST es pot visitar a través de visitors (un patró de disseny).
ANTLR també genera la interfície dels visitadors, tot generant un esquelet de mètodes que podem heretar. Cada mètode s'aplica sobre un tipus de node que correspon a cada regla de la gramàtica.

Anàlisi semàntica

L'analitzador semàntic recórre l'AST, per obtenir tota la informació necessària per poder generar codi. Objectius:

- Comprobar la correcció semàntica del programa (comprovació de tipus).
- Resoldre ambigüitats.
- Assignar memòria.
- Construir la taula de símbols.