Compilation

Langage de programmation	Langage machine
Haut niveau	Bas niveau
Syntaxe riche	Reflète l'architecture matérielle
Expressivité	Opérateurs explicites sur les unités fonctionnelles et la mémoire
structuration du programme = facilite le raisonnement	Organisation du programme adapté au matériel et qui fournis une bonne exploitation des ressources
existence de langages riches = dédiés à un domaine spécifique	

Vision plus générale

- Compilation = traduction et transformation d'un langage source vers un langage cible

Différents niveaux de langages

- langage machine = suite binaire interpelable par les micro-programmes du processeur
- langage assembleur = instructions représentées par des symboles mnémoniques
  - traduction en langage machine avec un assembleur
- langage de tout niveau
- langage intermédiaire = langage interne à un compilateur
  - commun à tous les langages sources pris en compte
  - la compilation effectue ses analyses et ses transformations sur la forme intermédiaire du programme
  - puis traduit la version finale en langage machine

Exemple :

- GCC = 3 formes intermédiaires
- CLANG
- bytecode Java ou .N ET

Différents types de compilateurs

- Interpréteur : le programme source est traduit en langage machine à la volée puis exécuté
  Exemple : python bash
- Machine virtuelle ou compilateur dynamique ou JIT (Just In Time compiler)
  - Prend en entrée un programme en langage intermédiaire, le traduit, l'optimise puis l'exécute
- Compilateur statique : prend en entrée un programme en langage source, le traduit, l'optimise et produit en sortie un programme en langage objet
  - compilateurs source-à-source
  - optimisation d'un programme au niveau du langage source-à-source

Niveau de compréhension d'un texte source
Texte source = suite de caractères

Premier niveau : reconnaissance de mots dans le texte
= groupes de caractères qui correspondent à un lexique
mots = unités syntaxiques, tokens
= Analyse lexicale

Deuxième niveau : trouver la structure du texte et vérifier s'il est conforme à une syntaxe
Syntaxe est définie pour une grammaire
= Analyse syntaxique

Troisième niveau

TD
Alphabet : Ensemble fini de symboles.
Mot : Séquence finie de symboles d'un alphabet.
Longueur d'un mot : Son nombre de symboles.
Langage : ensemble de mots sur un alphabet donné.
L'analyse lexicale cherche à déterminer le « statut » (= unité lexicale ~= token) qui lui correspond.

Donc étant donné un mot, quel est son statut ?
42 → entier littéral
3,14 → flottant littéral
« string » → chaine littérale
printf → identificateur
while → mot clé
; → punctuation

I.

1. [a-zA-Z_][a-zA-Z_0-9]*
2.
3. 0*10*10*10*
4. (1[01]*)? 0
5. [1-9][0-9]*
6. 0[0-7]
7. 0[xX][0-9a-fA-F]
8. 0[bB][01]+
9. ...[uU]?[lL]?[(ll)(LL)]?

Grammaires : composées d'un ensemble de productions mettant en jeu
- Les terminaux (unités lexicales)
- Les non-terminaux (variables syntaxiques)

Une production a la forme suivante :

Partie gauche (non-terminal)	→	Partie droite (liste de terminaux et non)

S → Aa
S → b
A → Ac
A → Sd
A → c

SA (NT) abc (T)
b, ca, bda, cca sont issues d'une suite de dérivations qui partent de l'axiome pour arriver vers des terminaux

bda= S → Aa → Sda → bda

Arbre d'analyse : f d'arbre d'une suite de dérivations
But de l'analyse syntaxique : trouver un arbre d'analyse étant donné une grammaire et un flux d'unités lexicales

étant donné une grammaire et un flux d'unités lexicales

- si l'arbre existe et est unique : tout va bien
- si l'arbre n'existe pas : syntax error
- s'il y a plusieurs arbres possibles : grammaire ambiguë

Analyse descendante
Les grammaires analysables de manière descendante sont dites LL
parsing Left to right and Leftmost derivation

Supporte pas les grammaires récursives à gauche ou factorisante à gauche

S → Sa S → ab
S → vide S → ac

Elimination de la récursivité à gauche :

- règle simple : A → Ax | y
  => A → yA'
  A' → xA' | vide
- cas des récursivités indirectes : substituer les NT de gauche jusqu'à ce qu'on ait plus de récursivité indirecte, alors voir ci-dessous : S → Aa | b
  A ->Ac | Sd | c
  => S → Aa | b
  A → Ac | <u>Aad | bd</u> | c
  => S → Aa | b
  A → <u>bdA'</u> | cA'
  A' → <u>adA'</u> | cA' | vide

Elimination de factorisation à gauche :

- règle simple : S → xA | xB
  => S-> xS'
  S' → A | B

Pour l'analyse on a besoin de deux fonctions :

- FIRST : Ensemble de terminaux qui peuvent apparaître en premier à partir d'un non-terminal
- FOLLOW : Ensemble de terminaux qui peuvent apparaître après un non-terminal

	U	*	(	)	id	e	$
E			E -> TE'		E -> TE'	E -> TE'	
E'	E' -> UTE'			E'-> &			
T			T -> FT'		T -> FT'	T -> FT'	
T'	T' → &	T' -> FT'	T' → &		T' -> FT'	T' -> FT'	T' → &
F			F → GG'		F → GG'	F → GG'	
G'	G' → &	G' → *	G' → &	G' → &	G' → &	G' → &	G' → &
G			G → (E)		G → id	G → e	

((Pile commence par axiome suivi de $))
Algorithme de reconnaissance : (avec a = symbole courant, X tete de pile et M la table)

Tant que X!= $
si X vaut a, dépiler et passer au token suivant
sinon si X est terminal : ERREUR
sinon si M[X, a] est vide : ERREUR
sinon si M[X, a] = X->Y1 Y2 .. Yn
Emettre la production
Dépile
Empile Yn .. Y2 Y1
(Y1 nouveau X)

S' -> S
S -> S(S) | &

X	First	Follow
S'	& (	$
S	& (	$ ( )

I0
S'->.S
S->.S(S)
S->.

Goto (I0, S) →

I1
S'->S.
S->S.(S)

Goto (I1, ( ) →

I2
S'->S(.S)
S->.S(S)
S->.

Goto (I2, S) →

I3
S'->S(S.)
S->S.(S)

Goto (I3, ( ) → I2
Goto (I3,)) →

I4
S'->S(S),

	(	)	$	S
I0	Reduce S->&	Reduce S->&	Reduce S->&	I1
I1	Shift I2		Accept	
I2	Reduce S->&	Reduce S->&	Reduce S->&	I3
I3	Shift I2	Shift I4		
I4	Reduce S->S(S)	Reduce S->S(S)	Reduce S->S(S)	
	Shift état/ Reduce règle/ Accept			Goto état

Construction : 3 cas pour tous les états

- point devant un terminal => shift
- point final => reduce
- S' => S. => accept

Pile	Chaine	Action
I0	(( ) ( ))$	Reduce → dépile |&| et empile goto (I0, S)
I0 I1	(( ) ( ))$	Shift → empile I2 et passe a
I0 I1 I2	( ) ( ))$	Reduce → dépile |&| et empile goto (I2, S)
I0 I1 I2 I3	( ) ( ))$	Shift → empile I2 et passe a
I0 I1 I2 I3 I2	) ( ))$	Reduce → dépile |&| et empile goto (I2, S)
I0 I1 I2 I3 I2 I3	) ( ))$	Shift → empile I4 et passe a
I0 I1 I2 I3 I2 I3 I4	( ))$	Reduce → dépile |S(S)| et empile goto (I2, S)
I0 I1 I2 I3	( ))$	Shift → empile I2 et passe a
I0 I1 I2 I3 I2	))$	Reduce → dépile |&| et empile goto (I2, S)
I0 I1 I2 I3 I2 I3	)$	Shift → empile I4 et passe a
I0 I1 I2 I3 I2 I3 I4	)$	Reduce → dépile |S(S)| et empile goto (I2, S)
I0 I1 I2 I3	)$	Shift → empile I4 et passe a
I0	$	Reduce → dépile |S(S)| et empile goto (I0, S)
I0 I1	$	Accept