Polycopié de CS444

— Version 2022/2023 —

Laure Gonnord - C. Deleuze. (et M. Moy, etc, Lyon1)

Premature optimization is the root of all evil (or at least most of it) in programming.

Donald Knuth, *Décembre 1974, Conférence du Prix Turing 1974, Communications of the ACM.*

Afin d'améliorer ce poly n'hésitez pas à me soumettre toute critique, suggestion, remarque ou correction, dans mon casier ou, électroniquement, à l'adresse

`Laure.Gonnord@esisar.grenoble-inp.fr`

Table des matières

Compilation (#1a) – Aperçu

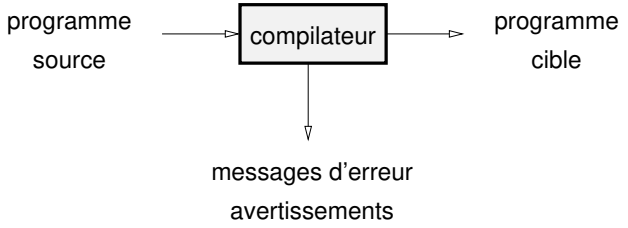
C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

Plan

Compiler = ?



Exemple

```
int fact(int n)
{
 if (n==0) return 1;
 else return(n*fact(n-1));
}
```

```
$ gcc -m32 -S fact.c
```

```
fact:
        pushl %ebp
        movl %esp, %ebp
        subl $24, %esp
        cmpl $0, 8(%ebp)
        jne .L2
        movl $1, %eax
        jmp .L3
.L2:
        movl 8(%ebp), %eax
        subl $1, %eax
        movl %eax, (%esp)
        call fact
        imull 8(%ebp), %eax
.L3:
        leave
        ret
```

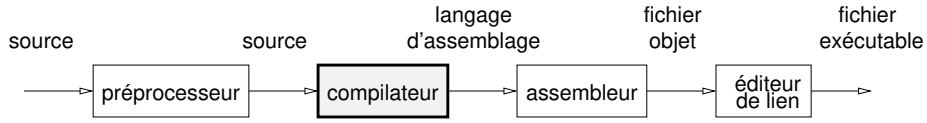

Pourquoi étudier la compilation ?

- pb compliqué, bien résolu
  - structuration du pb
  - utilisation judicieuse de formalismes
  - utilisation d'outils
- touche à presque tout
  - de la manipulation de texte,
  - aux architectures de machines,
  - en passant par les concepts des langages de programmation

Pourquoi étudier la compilation ?

- pb compliqué, bien résolu
  - structuration du pb
  - utilisation judicieuse de formalismes
  - utilisation d'outils
- touche à presque tout
  - de la manipulation de texte,
  - aux architectures de machines,
  - en passant par les concepts des langages de programmation
- la compilation c'est "cool" !

La chaine de compilation



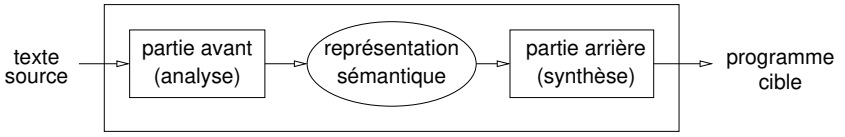
Un programme est . . . une suite de caractères ?

. . . organisés suivant une certaine structure

- structure lexicale
- structure syntaxique
- structure sémantique/contextuelle

```
position := initiale + vitesse * 60
```


Analyse et synthèse

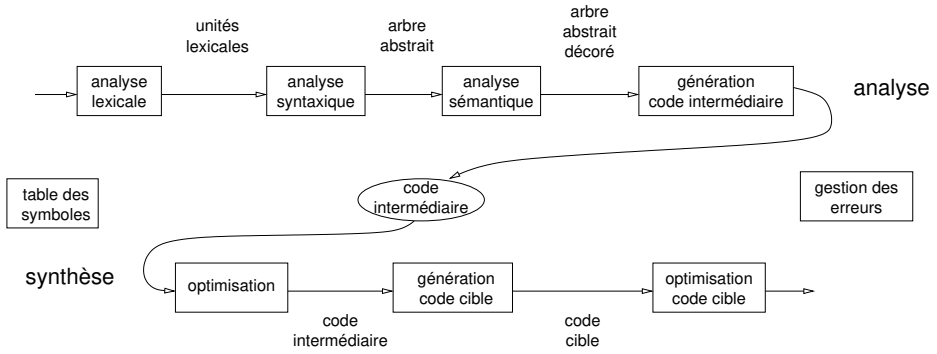


Analyse et synthèse

analyse à partir de la représentation concrète d'un pgm, produire une forme abstraite mettante en évidence sa structure et donc sa sémantique

synthèse à partir d'une représentation abstraite de la sémantique, produire une représentation dans la syntaxe concrète du langage cible

Phases



Analyse lexicale

mise en oeuvre par le "lexer"

- entrée : flot de caractères
- sortie : flot d'unités lexicales
- élimine blancs et commentaires
- entre les identificateurs dans la table des symboles

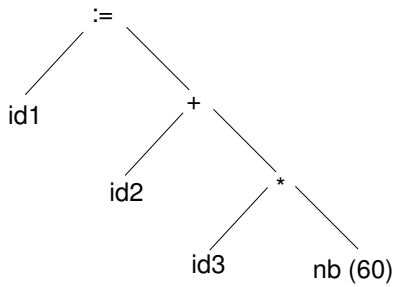
position := initiale + vitesse * 60

$id_1 := id_2 + id_3 * nb_{60}$

Analyse syntaxique

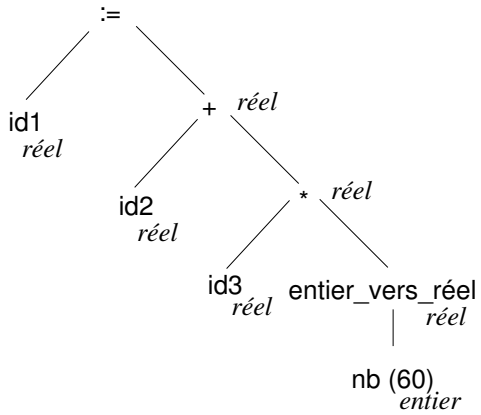
mise en oeuvre par le "parser"

- groupe les UL en structures syntaxiques
- produit un "arbre abstrait"



Analyse sémantique

- collecte les infos de type
  - pour recherche d'erreurs sémantiques
  - et pour usage par les phases suivantes
- "décore" l'arbre abstrait avec des infos de contexte



Génération de code intermédiaire

ex : code à trois adresses
langage d'assemblage pour une machine abstraite

- chaque instruction a trois opérandes (max)
- nombre infini de registres

```
t1 := entierVersRéel(60)
t2 := id3 *r t1
t3 := id2 +r t2
id1 := t3
```


Optimisation de code intermédiaire

optimiser = ?

```
t1 := id3 *r 60.0
id1 := id2 +r t1
```

Code cible

Génération de code cible

- sélection des emplacements mémoire
- traduction de chaque inst CI en séquence d'insts ASM
  - assignation des registres

Optimisation de code cible

- liées à l'architecture et au jeu d'intructions de la machine cible

Table des symboles

utilisée par toutes les phases
la table contient tous les identificateurs du pgm avec les infos les concernant :

- type
- portée
- classe
- adresse
- ...

Gestion des erreurs

Dans les phases d'analyse :

- erreur lexicale
- erreur syntaxique
- erreur sémantique

difficultés :

- donner un message approprié sur l'emplacement et la cause probable de l'erreur
- "récupération" : corriger l'erreur pour continuer l'analyse

Phases et passes

phase une étape conceptuelle dans le travail de compilation

passe parcours de l'ensemble du programme, effectuant un traitement

les trois phases d'analyse lexicale, syntaxique et sémantique sont souvent regroupées dans une seule passe

Plan du cours

séances :

1. aperçu, analyse lexicale
2. analyse syntaxique
3. analyse syntaxique
4. grammaires attribuées
5. arbre abstrait, typage
6. génération de code
7. représentations intermédiaires
8. allocation de registres
9. analyse de flot de données

Bilan

Compilation (#2) : Analyse lexicale

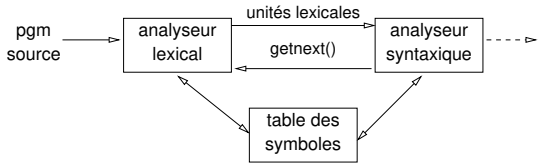
C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

Plan

Place de l'analyseur lexical



UNITÉ LEXICALE	LEXÈMES	DESCRIPTION INFORMELLE DES MODÈLES
if	`if`	`if`
then	`then`	`then`
assign	`:=`	`:=`
oprel	`< <= > >= = <>`	`< <= > >= = <>`
id	`position vitesse ...`	lettre suivie de lettres ou chiffres
nb	`3.1416 0 6.02E23 ...`	toute constante numérique
chaîne	`"core dumped" ...`	tous caractères entre " et " sauf "

Figure: Exemples d'unités lexicales

Attributs des unités lexicales

...

- alphabet, mot (on dira chaîne)

Spécification des unités lexicales Chaines et langages

- alphabet, mot (on dira chaîne)
- langage

Deleuze, Gonnord (Esisar) Compilation #1b Analyse lexicale 2022-2023 ← 9 / 31 →

Spécification des unités lexicales Chaines et langages

- alphabet, mot (on dira chaîne)
- langage
  - opérations sur les langages

Deleuze, Gonnord (Esisar) Compilation #1b Analyse lexicale 2022-2023 ← 9 / 31 →

Spécification des unités lexicales Chaines et langages

- alphabet, mot (on dira chaîne)
- langage
  - opérations sur les langages

OPÉRATION	NOTATION	DÉFINITION
union de L et M	L ∪ M	$\{s \mid s \in L \text{ ou } s \in M\}$
concaténation de L et M	LM	$\{st \mid s \in L \text{ et } t \in M\}$
fermeture positive	L^+	$\cup_{i=1}^{\infty} L^i$
fermeture de Kleene	L^*	$\cup_{i=0}^{\infty} L^i$

Figure: Définitions des opérations sur langages

Deleuze, Gonnord (Esisar) Compilation #1b Analyse lexicale 2022-2023 ← 9 / 31 →

2 Spécification des unités lexicales
- Chaines et langages
- Expressions régulières

Règles de définition des expressions régulières sur un alphabet Σ

- chapitre 1 du cours CS322
  - on notera | au lieu de $+$

Règles de définition des expressions régulières sur un alphabet Σ

- chapitre 1 du cours CS322
  - on notera | au lieu de $+$

1. ϵ est l'expression régulière qui dénote $\{\epsilon\}$
2. $a \in \Sigma$ a = expression régulière qui dénote $\{a\}$
3. r et s expressions régulières qui dénotent L(r) et L(s)

   r | s dénote L(r) $\cup$ L(s)

   rs dénote L(r)L(s)

   r* dénote (L(r))*

1 Rôle de l'analyseur lexical

2 Spécification des unités lexicales

3 Reconnaissance des unités lexicales
- Automates finis
- Construction d'un analyseur lexical

4 Outils de construction d'analyseurs lexicaux

Un reconnaisseur pour un langage est un programme qui prend en entrée une chaîne x et répond si oui ou non x est une chaîne du langage.

Automate fini non déterministe

C'est un modèle mathématique qui consiste en :

1. ensemble d'états E
2. ensemble de symboles d'entrée Σ (alphabet des symboles d'entrée)
3. fonction de transition Transiter : (état, symbole) $\rightarrow$ { états } (avec symbole $\in \Sigma \cup \{\epsilon\}$)
4. état distingué "initial"
5. ensemble d'états distingués "finaux"

3 Reconnaissance des unités lexicales
- Automates finis
- Construction d'un analyseur lexical

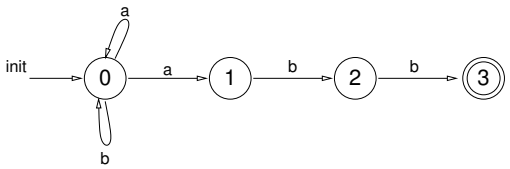
Automate fini non déterministe

C'est un modèle mathématique qui consiste en :

1. ensemble d'états E
2. ensemble de symboles d'entrée Σ (alphabet des symboles d'entrée)
3. fonction de transition Transiter : (état, symbole) $\rightarrow$ { états } (avec symbole $\in \Sigma \cup \{\epsilon\}$)
4. état distingué "initial"
5. ensemble d'états distingués "finaux"

Un AFN accepte une chaîne x s'il existe un chemin dans le graphe entre l'état initial et un état final tel que les étiquettes d'arcs le long de ce chemin épellent le mot x.

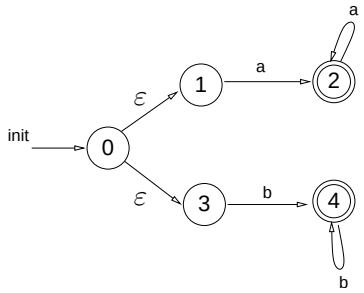
AFN reconnaissant le langage (a|b)*abb



état	symbole d'entrée		
	a	b	ϵ
0	{ 0, 1 }	{ 0 }	-
1	-	{ 2 }	-
2	-	{ 3 }	-

Figure: Table de transition de l'automate précédent

Un autre AFN

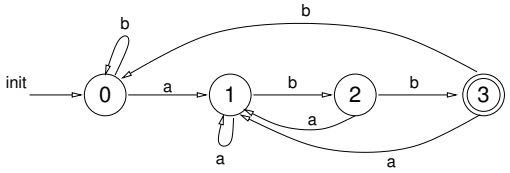


Automates finis déterministes (AFD)

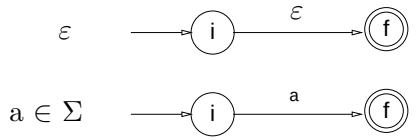
Cas particulier d'AFN.

1. aucune ϵ-transition,
2. pour chaque état e et symbole a, au plus un arc étiqueté a quittant e.

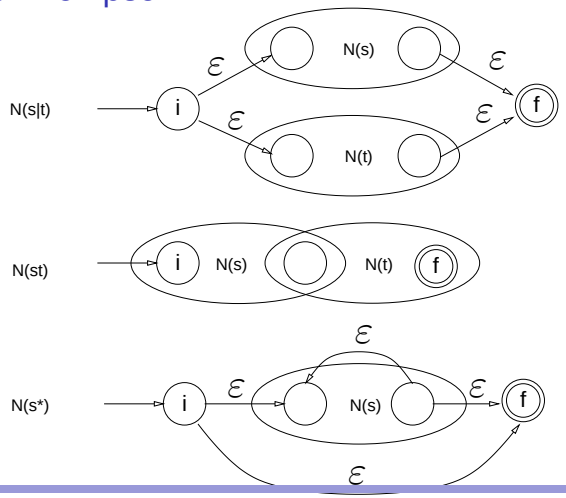
AFD pour le langage (a|b)*abb



Construction de Thompson – 1



Construction de Thompson – 2

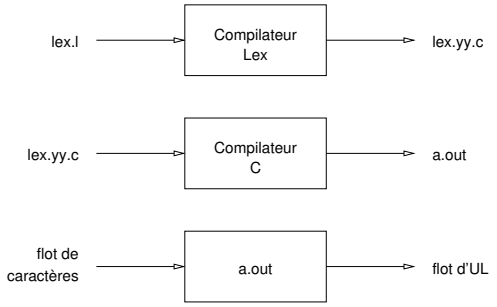


- construire un automate pour chaque modèle
- les fusionner (en distinguant les états finaux)
- rendre déterministe/minimiser
- implémenter/exécuter l'automate

- construire un automate pour chaque modèle
- les fusionner (en distinguant les états finaux)
- rendre déterministe/minimiser
- implémenter/exécuter l'automate

une petite subtilité (voir postparation)

Utilisation de lex



Spécification lex

en trois parties

déclarations *eg macros pour les regexps*

%%

règles de traduction

m_1 { action_1 }
m_2 { action_2 }
...
m_n { action_n }

%%

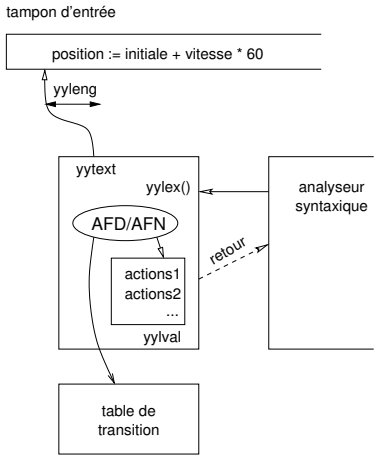
procédures auxiliaires *utilisées dans les actions*

m_i = modèle

action_i = code à exécuter quand un lexème concorde avec le modèle


```
/* table de transition
   insérée ici ... */

int yylex() {
  while(1) {
    simule automate jusqu'à match

    fixe yytext   /* le lexème */
         yyleng   /* trouvé    */

    if m_1 actions_1  /* typique : return(UL) */
    if m_2 actions_2  /* peut fixer yylval pour attribut */
    ...
  }
`
```


Fonctionnement du code généré par lex



Postparation

Soient les unités lexicales et modèles suivants :

id	lettre suivie de lettres ou chiffres
assign	=
oprel	< <= > >=

On a construit un automate reconnaissant chacun des trois modèles, on les a fusionnés et on analyse lexicalement le texte :

a <= b

Quelles unités lexicales vont être reconnues ? Comment résoudre le problème ?

Bilan

1. Rôle de l'analyseur lexical
2. Spécification des unités lexicales
   - Chaines et langages
   - Expressions régulières
3. Reconnaissance des unités lexicales
   - Automates finis
   - Construction d'un analyseur lexical
4. Outils de construction d'analyseurs lexicaux

Compilation (#3) : Analyse syntaxique

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

Plan

Définition informelle de la syntaxe

une instruction est définie par :

1. si *id* est un identificateur et *exp* une expression alors
   id := *exp* est une instruction
2. si *exp* est une expression et *inst* une instruction alors

si *exp* alors *inst* tant que *exp* faire *inst*	sont des instructions

Définition informelle de la syntaxe

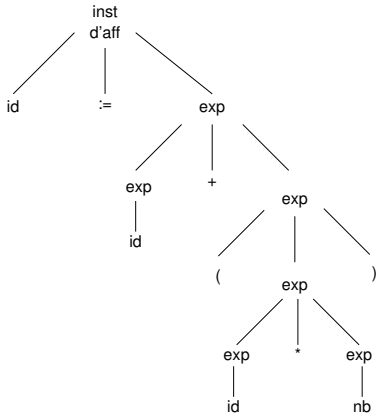
une instruction est définie par :

1. si *id* est un identificateur et *exp* une expression alors
   id := *exp* est une instruction
2. si *exp* est une expression et *inst* une instruction alors
   si *exp* alors *inst*
   tant que *exp* faire *inst*
   | sont des instructions

une expression est définie par :

3. tout identificateur est une expression
4. tout nombre est une expression
5. si *exp1* et *exp2* sont des expressions alors
   exp1 + *exp2*
   exp1 * *exp2*
   (*exp1*)
   | sont des expressions

Arbre syntaxique



Grammaires

La syntaxe des langages de programmation est définie à l'aide d'un formalisme appelé grammaire non contextuelle (ou simplement grammaire).
Une grammaire comprend quatre composants :

1. un ensemble d'unités lexicales, appelées symboles terminaux,
2. un ensemble de symboles non terminaux,
3. un ensemble de productions,

   non terminal → suite de terminaux et non terminaux

   partie gauche — *partie droite* (éventuellement ϵ)
4. un non terminal est désigné comme **axiome**.

Voici la grammaire décrite informellement ci-dessus.

I → **id** := E
I → **si** E **alors** I
I → **tant que** E **faire** I

E → **id**
E → **nb**
E → E + E
E → E * E
E → (E)

Quelques définitions

- Une production **définit** un non terminal s'il apparaît en partie gauche de la production.
- Une **chaîne** d'unités lexicales est une suite éventuellement vide d'unités lexicales (notée ϵ si vide).
- Une grammaire **dérive** des chaînes en commençant par l'axiome et en remplaçant, de manière répétée, un non terminal par la partie droite d'une des productions le définissant.
- Les chaînes d'unités lexicales que l'on peut dériver ainsi constituent le **langage défini** par la grammaire.

Dérivation

Arbre d'analyse

C'est la représentation graphique d'une dérivation dans laquelle les choix concernant l'ordre de dérivation ont disparu.

- chaque nœud intérieur est étiqueté par un non terminal,
- ses fils sont étiquetés par les symboles de la partie droite de la production utilisée,
- les feuilles forment la chaîne des symboles terminaux (de gauche à droite).

Arbre d'analyse

C'est la représentation graphique d'une dérivation dans laquelle les choix concernant l'ordre de dérivation ont disparu.

- chaque nœud intérieur est étiqueté par un non terminal,
- ses fils sont étiquetés par les symboles de la partie droite de la production utilisée,
- les feuilles forment la chaîne des symboles terminaux (de gauche à droite).

aka arbre syntaxique !

Ambiguïtés

Non aux ambiguïtés !

Grammaire non ambigüe :
à chaque texte de programme correspond :

Non aux ambiguïtés !

Grammaire non ambigüe :
à chaque texte de programme correspond :

- soit un unique arbre de dérivation et le programme est (syntaxiquement) correct,
- soit aucun et le pgm contient au moins une erreur syntaxique.

Non aux ambiguïtés !

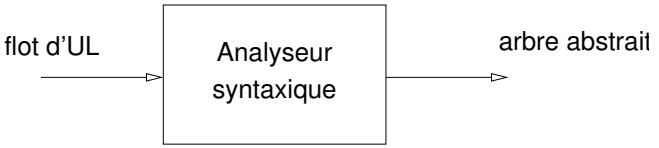
Grammaire non ambigüe :
à chaque texte de programme correspond :

- soit un unique arbre de dérivation et le programme est (syntaxiquement) correct,
- soit aucun et le pgm contient au moins une erreur syntaxique.

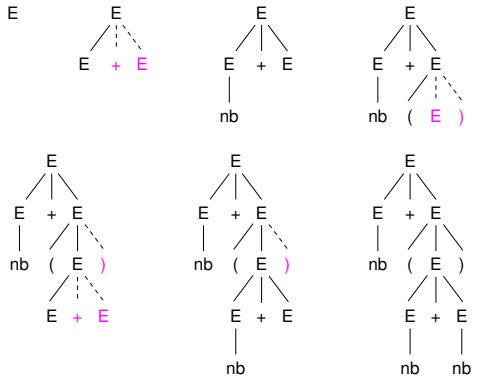
Deux façons d'éliminer les ambiguïtés :

- modifier la grammaire pour la rendre non ambiguë,
- lever les ambiguïtés avec des règles externes à la grammaire.

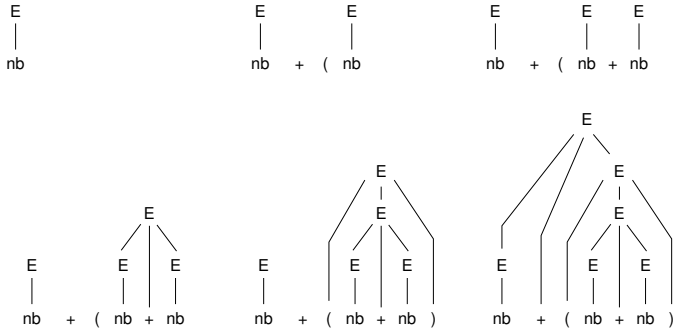
Analyseur syntaxique



Analyse descendante



Analyse ascendante



expression → terme reste_expression	E → T R
terme → expression_parenthésée | **id**	T → P | id
expression_parenthésée → '(' expression ')'	P → (E)
reste_expression → '+' expression | ε	R → + E | ε

Figure: Grammaire pour montrer l'analyse descendante

1. $\forall t$ terminal, PREM$(t) = \{t\}$
2. $\forall A$ non-terminal, PREM$(A) = \emptyset$
3. Pour les non terminaux, appliquer les règles suivantes jusqu'à ce qu'on ne puisse plus rien ajouter (A est un non terminal, X_i un symbole quelconque) :
   1. si $A \to X_1 X_2 ... X_k$ avec $k \geqslant 1$
      - ajouter PREM$(X_1)\backslash\{\varepsilon\}$ à PREM(A)
      - si $\varepsilon \in$ PREM(X_1), ajouter PREM$(X_2)\backslash\{\varepsilon\}$
      - si en plus, $\varepsilon \in$ PREM(X_2), ajouter PREM$(X_3)\backslash\{\varepsilon\}$
      - ...
      - si $\varepsilon \in$ PREM(X_i) $\forall i$, ajouter ε à PREM(A).
   2. si $A \to \varepsilon$ ajouter ε à PREM(A).

Figure: Calcul des PREMs des symboles

pour une chaîne $\alpha = X_1 X_2 \dots X_n$, avec X_i un symbole quelconque

1. initialiser PREM(α) à $\emptyset$
2. ajouter PREM(X_1)$\backslash\{\varepsilon\}$
3. si $\varepsilon \in$ PREM(X_1), ajouter PREM(X_2)$\backslash\{\varepsilon\}$
4. si en plus $\varepsilon \in$ PREM(X_2), ajouter PREM(X_3)$\backslash\{\varepsilon\}$
5. ...
6. si $\varepsilon \in$ PREM(X_i) $\forall i$, ajouter ε

Figure: Calcul des PREMs des chaînes

Initialiser le SUIV de l'axiome à {$} et les autres SUIV à $\emptyset$ puis appliquer les règles suivantes jusqu'à ce qu'on ne puisse plus rien ajouter :

1. pour chaque production de la forme $A \to \alpha B \beta$
   - ajouter PREM(β)$\backslash\{\varepsilon\}$ à SUIV(B)
   - si $\varepsilon \in$ PREM(β), ajouter SUIV(A) à SUIV(B)
2. pour chaque production de la forme $A \to \alpha B$
   - ajouter SUIV(A) à SUIV(B).

Figure: Calcul des SUIVs des non terminaux

Non terminal	PREM	SUIV
expression	{ id '(' }	{ $ ')' }
terme	{ id '(' }	{ '+' $ ')' }
expression_parenthésée	{ '(' }	{ '+' $ ')' }
reste_expression	{ '+' ε }	{ $ ')' }

Figure: Ensembles PREM et SUIV de notre grammaire


```
#include "ana_lex.h"
#include "ap.h"

void parse(void) {
  expression();
  consomme_ul(FDF);
}

void expression(void) {
  switch(prochaine_ul()) {
  case ID:
  case '(':  terme(); reste_expression(); break;
  default:   erreur();
  }
```

```
}

void terme(void) {
  switch(prochaine_ul()) {
  case '(':  expression_parenthesee(); break;
  case ID:   consomme_ul(ID); break;
  default:   erreur();
  }
}

void expression_parenthesee(void) {
  switch(prochaine_ul()) {
  case '(':  consomme_ul('('); expression(); consomme_ul(')'); break;
  default:   erreur();
  }
```



```
}

void reste_expression(void) {
  switch(prochaine_ul()) {
  case '+':  consomme_ul('+'); expression(); break;
  case FDF:
  case ')':  break;
  default:   erreur();
  }
}
```

Figure: Analyseur prédictif


```
void terme(void) {
    switch(prochaine_ul()) {
    case '(': expression_parenthesee(); break;
    case ID:  consomme_ul(ID); break;
    case ID:  element_indexe(); break;
    default:  erreur();
    }
}
```

Figure: Code erroné pour l'analyse d'un terme avec `élément_indexé`

```
void S(void) {
    switch(prochaine_ul()) {
    case 'a': A(); consomme_ul('a'); consomme_ul('b'); break;
    default:  erreur();
    }
}

void A(void) {
    switch(prochaine_ul()) {
    case 'a': consomme_ul('a'); break;
    case 'a': break;
    default:  erreur();
    }
}
```

Figure: Code erroné avec un conflit PREM-SUIV

Conditions pour être LL(1)

- pas de conflit PREM-PREM : $\forall$ N, PREM de tous les choix sont distincts,
- pas de conflit PREM-SUIV : $\forall$ N nullifiable, SUIV(N) distinct des PREM des autres choix,
- pas de choix nullifiables multiples.

non terminal	terminal en entrée				
	id	+	(	)	$
expression	terme rest_exp		terme rest_exp		
terme	id		exp_par		
exp_par			(expression)		
rest_exp		+ expression		ε	ε

Figure: Table de transition de l'analyseur prédictif itératif

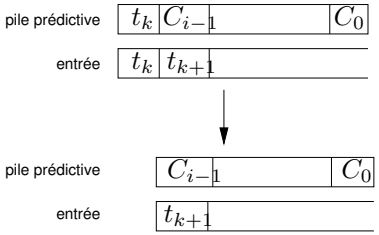
Mouvements de l'analyseur

reconnaissance Un terminal est en sommet de pile. Il doit être égal au terminal en entrée. On dépile et avance (ou erreur).

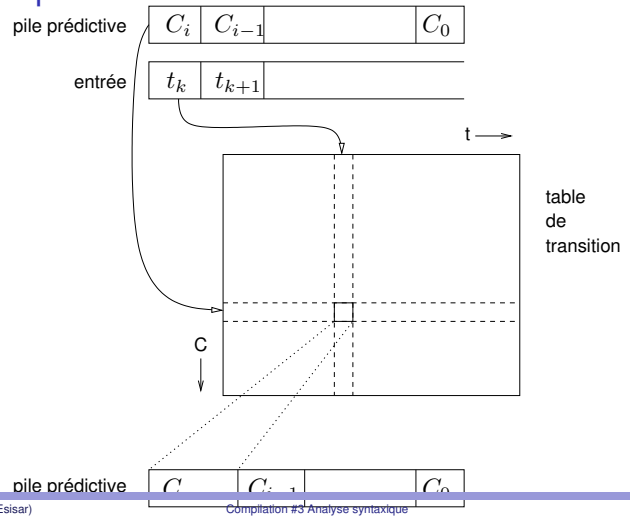
prédiction Un non-terminal est en sommet de pile. Dépile, consulte table et empile les nouveaux symboles (ou erreur).

terminaison La pile prédictive est vide.

Mouvement de reconnaissance



Mouvement de prédiction




```
importer Entrée [1..]  // depuis l'analyseur lexical
Indice_entrée ← 1
Pile_prédictive ← Pile_vide
empiler FDF puis axiome sur Pile_prédictive
tantque Pile_prédictive ≠ Pile_vide faire
    prédiction ← dépiler(Pile_prédictive)

    si prédiction est un terminal alors
        // essayer mouvement de reconnaissance
        si prédiction = Entrée[Indice_entrée] alors
                Indice_entrée ← Indice_entrée + 1
        sinon
                erreur "UL attendue non trouvée : ", prédiction
    sinon
        // essayer mouvement de prédiction
        prédiction ← table_prédictive[prédiction, Entrée[Indice_entrée]]
        si prédiction = vide alors
                erreur "UL inattendue : ", Entrée[Indice_entrée]
        sinon
                empiler les symboles de prédiction sur Pile_prédictive
```

Exercice : analyse de `id + (id + id)`

NT	terminal				
	id	+	(	)	$
E	T R		T R		
T	id		P		
P			(E)		
R		+ E		ε	ε

Figure: Table de transition de l'analyseur prédictif itératif

Postparation

1. Ambiguïtés : montrez deux dérivations de la chaîne `id := id + id * nb` qui correspondent à des arbres syntaxiques différents (grammaire du transparent 7).
2. Exécutez à la main l'analyseur prédictif récursif sur la chaîne `id + id`, puis sur la chaîne `id + (id + id)`.
3. Soit une grammaire contenant la production E $\rightarrow$ E + E. Expliquez pourquoi elle n'est pas LL(1).
4. Terminez l'analyse prédictive itérative du transparent 37,
   - et "retrouvez" la dérivation correpondante dans l'analyse.

Préparation

Lisez l'interview du créateur de Yacc, dans le document "The A-Z of Programming Languages" (document sur chamilo)

Pivot

Le but est de trouver le nœud pas encore construit le plus à gauche dont tous les fils ont été construits. Cette suite de fils constitue le *pivot*. Créer le nœud N du parent et le relier aux fils (le pivot), c'est *réduire* le pivot vers N.

Définition

On appelle *pivot* (ou *manche* ou *poignée* – *handle* en anglais) une suite β de symboles terminaux et non terminaux qui correspond à la partie droite d'une production et dont la réduction vers le non terminal de la partie gauche de cette production représente une étape le long de la dérivation droite inverse.

Réduction du pivot

Soit une chaîne de terminaux w, appartenant au langage.
Il existe une dérivation droite (inconnue) :

$$S = \gamma_0 \underset{d}{\rightarrow} \gamma_1 \underset{d}{\rightarrow} \gamma_2 \quad \dots \quad \underset{d}{\rightarrow} \gamma_{n-1} \underset{d}{\rightarrow} \gamma_n = w$$

Repérer le pivot β_n dans γ_n et le remplacer par le A de la production A $\rightarrow \beta_n$ donne γ_{n-1}...
repeat

Actions de l'analyseur

décaler prochain terminal de l'entrée est retiré et empilé.

réduire le pivot est en sommet de pile. Il est remplacé par la partie gauche de la production choisie.

accepter la pile contient uniquement l'axiome et l'entrée est vide.

erreur

1,2	expression → terme | expression '+' terme	E → T | E+T
3,4	terme → **id** | '(' expression ')'	T → id | (E)

Figure: Une grammaire récursive à gauche

Pile	Entrée	Action
	(a+b)+c	

1,2	E → T | E+T
3,4	T → id | (E)

Pile	Entrée	Action
	(a+b)+c	D

1,2	E → T | E+T
3,4	T → id | (E)

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Analyse ascendante Analyse par décalage-réduction

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Deleuze, Gonnord (Esisar) Compilation #3 Analyse syntaxique 2022-2023 ← 47 / 72 →

Analyse ascendante Analyse par décalage-réduction

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Deleuze, Gonnord (Esisar) Compilation #3 Analyse syntaxique 2022-2023 ← 47 / 72 →

Analyse ascendante Analyse par décalage-réduction

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Deleuze, Gonnord (Esisar) Compilation #3 Analyse syntaxique 2022-2023 ← 47 / 72 →

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Analyse ascendante Analyse par décalage-réduction

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	D
(E+b	)+c	

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Deleuze, Gonnord (Esisar) Compilation #3 Analyse syntaxique 2022-2023 ← 47 / 72 →

Analyse ascendante Analyse par décalage-réduction

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	D
(E+b	)+c	R3

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Deleuze, Gonnord (Esisar) Compilation #3 Analyse syntaxique 2022-2023 ← 47 / 72 →

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

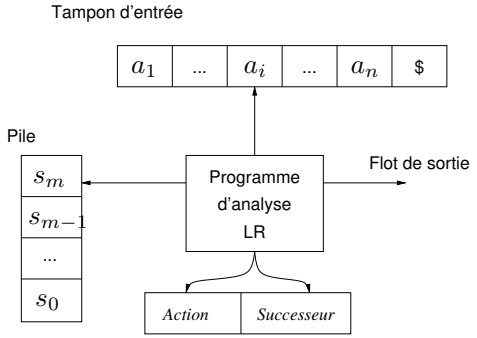
Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	D
(E+b	)+c	R3
(E+T	)+c	

1,2 $E \rightarrow T \mid E+T$
3,4 $T \rightarrow id \mid (E)$

Pile	Entrée	Action
	(a+b)+c	D
(	a+b)+c	D
(a	+b)+c	R3
(T	+b)+c	R1
(E	+b)+c	D
(E+	b)+c	D
(E+b	)+c	R3
(E+T	)+c	R2

LR(k)
méthode décalage-réduction sans rebroussement *(backtracking)* la plus générale connue.
le but est de **repérer** le pivot
inconvénient : trop de travail pour être mise en œuvre manuellement sur la grammaire d'un vrai langage de programmation

Modèle d'un analyseur LR



Actions

configuration de l'analyseur

(s_0 s_1 ... s_m , a_i a_{i+1} ... a_n $)

Action[s_m, a_i] = décaler s

(s_0 s_1 ... s_m s , a_{i+1} ... a_n $)

Action[s_m, a_i] = réduire par A $\rightarrow \beta$

(s_0 s_1 ... s_{m-r} s, a_i a_{i+1} ... a_n $)

- r = | β |
- s = Successeur[s_{m-r}, A]

Algorithme LR

```
soit a le premier symbole de l'entrée
empiler s0
accepter ← faux
erreur ← faux
répéter
    soit s l'état en sommet de pile
    si Action[s,a] = décaler s' alors
        empiler s'
        soit a le symbole d'entrée suivant
    sinon si Action[s,a] = réduire par A → β alors
        dépiler |β| états
        soit s' l'état en sommet de pile
        empiler Successeur[s',A]
    sinon si Action[s,a] = accepter alors accepter ← vrai
    sinon erreur ← vrai
```


Une grammaire

1 E → E + T
2 E → T
3 T → T * F
4 T → F
5 F → (E)
6 F → **id**

Table d'analyse pour la grammaire

État	Action						Successeur		
	id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	d3		r3	r3			
11		r5	r5		r5	r5			

Table d'analyse pour la grammaire

1 $E \rightarrow E + T$
2 $E \rightarrow T$
3 $T \rightarrow T * F$
4 $T \rightarrow F$
5 $F \rightarrow (E)$
6 $F \rightarrow$ **id**

État	Action						Successeur		
	id	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				acc			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	d3		r3	r3			
11		r5	r5		r5	r5			

Pile états	symboles (pour info)	Entrée	Action
0		id*id+id $	d5
0 5	id	*id+id $	r6
0 3	F	*id+id $	r4
0 2	T	*id+id $	d7
0 2 7	T*	id+id $	d5
0 2 7 5	T*id	+id $	r6
0 2 7 10	T*F	+id $	r3
0 2	T	+id $	r2
0 1	E	+id $	d6
0 1 6	E +	id $	d5
0 1 6 5	E + id	$	r6
0 1 6 3	E + F	$	r4
0 1 6 9	E + T	$	r1
0 1	E	$	acc

Grammaire LR

- Une grammaire LR est une grammaire pour laquelle il est possible de construire la table d'analyse.
- Pour qu'une grammaire soit LL(k) il faut pouvoir reconnaître l'usage d'une production à la vue des k premiers symboles dérivés de sa partie droite.
- Pour qu'une grammaire soit LR(k) il faut pouvoir reconnaître le pivot (partie droite de production) en ayant vu tout ce qui est dérivé de cette partie droite et à la vue des k prochains symboles de l'entrée.

Grammaire LR

- Une grammaire LR est une grammaire pour laquelle il est possible de construire la table d'analyse.
- Pour qu'une grammaire soit LL(k) il faut pouvoir reconnaître l'usage d'une production à la vue des k premiers symboles dérivés de sa partie droite.
- Pour qu'une grammaire soit LR(k) il faut pouvoir reconnaître le pivot (partie droite de production) en ayant vu tout ce qui est dérivé de cette partie droite et à la vue des k prochains symboles de l'entrée.
- donc LL(k) .... LR(k)

Items LR(0)

item d'une grammaire G = production de G avec un point repérant une position dans la partie droite.

- A -> XYZ fournit 4 items :
  - A -> •XYZ
  - A -> X•YZ
  - A -> XY•Z
  - A -> XYZ•
- A -> ε fournit A -> •

Opération *Fermeture*

I un ensemble d'items pour une grammaire G
Fermeture(I) est l'ensemble d'items construit à partir de I par les deux règles :

1. Initialement, placer chaque item de I dans *Fermeture*(I).
2. Si $A \to \alpha \bullet B\beta$ est dans *Fermeture*(I),
   pour chaque production $B \to \gamma$ de G,
   ajouter l'item $B \to \bullet\gamma$ à *Fermeture*(I) (si pas déjà présent).

   Appliquer cette règle jusqu'à ce que plus aucun item ne puisse être ajouté à *Fermeture*(I).

Opération *Transition*

Transition(I,X)

- I un ensemble d'items
- X symbole de la grammaire

Transition(I,X) est la fermeture de l'ensemble de tous les items $A \to \alpha X \bullet \beta$ tels que $A \to \alpha \bullet X\beta$ appartienne à I.

Construction des ensembles d'items

```
C ← { Fermeture({E'→ • E}) }
répéter
    pour chaque ensemble d'items I de C et
    pour chaque symbole X de la grammaire tel que
                Transition(I,X) soit non vide
                et non encore dans C faire
         ajouter Transition(I,X) à C
jusqu'à ce que plus aucun nouvel ensemble d'items
        ne puisse être ajouté à C
```


	E' → E
1,2	E → E + T | T
3,4	T → T * F | F
5,6	F → (E) | **id**

```
C ← { Fermeture({E'→ • E}) }
répéter
    pour chaque ensemble d'items I de C et
    pour chaque symbole X de la grammaire tel que
                Transition(I,X) soit non vide
                et non encore dans C faire
         ajouter Transition(I,X) à C
jusqu'à ce que plus aucun nouvel ensemble d'items
        ne puisse être ajouté à C
```

Construction des tables d'analyse SLR

1. Construire C = $\{I_0, I_1, ..., I_n\}$, la collection des ensembles d'items LR(0) pour G'.
2. L'état i est construit à partir de I_i. Les actions d'analyse pour l'état i sont déterminées comme suit :
   1. Si $A \to \alpha \bullet a\beta$ est dans I_i et *Transition(I_i,a)* = I_j, remplir *Action[i,a]* avec *"décaler j"* (a est un terminal).
   2. Si $A \to \alpha\bullet$ est dans I_i, remplir *Action[i,a]* avec *"réduire par* $A \to \alpha$*"* pour tous les SUIV(A) (A ne doit pas être E').
   3. si $E' \to E\bullet$ est dans I_i, remplir *Action[i,$]* avec *"accepter"*.

   Si les règles précédentes engendrent des actions conflictuelles, la grammaire n'est pas SLR(1).
3. si *Transition*(I_i,A) = I_j alors *Successeur*[i,A] = j.
4. Toutes les entrées non définies sont positionnées à *"erreur"*.

E' → E
1,2 E → E + T | T
3,4 T → T * F | F
5,6 F → (E) | **id**

1. Construire C = $\{I_0, I_1, ..., I_n\}$, la collection des ensembles d'items LR(0) pour G'.
2. L'état i est construit à partir de I_i. Les actions d'analyse pour l'état i sont déterminées comme suit :
   1. Si $A \to \alpha \bullet a\beta$ est dans I_i et *Transition(I_i,a)* = I_j, remplir *Action[i,a]* avec *"décaler j"* (a est un terminal).
   2. Si $A \to \alpha\bullet$ est dans I_i, remplir *Action[i,a]* avec *"réduire par* $A \to \alpha$*"* pour tous les SUIV(A) (A ne doit pas être E').
   3. si $E' \to E\bullet$ est dans I_i, remplir *Action[i,$]* avec *"accepter"*.
3. si *Transition*(I_i,A) = I_j alors *Successeur*[i,A] = j.

État	Action						Successeur		
	id	+	*	(	)	$	E	T	F
0									
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									

Comment ça marche ?

Les principales idées...

- proto-phrases droites, préfixes viables
- on a construit l'**automate LR(0)** qui reconnait les **préfixes viables** de la grammaire.
- notion d'**items valides** pour un préfixe viable
- L'ensemble des items atteint depuis l'état initial le long du chemin étiqueté γ dans l'automate LR(0) de la grammaire *est l'ensemble des items valides pour le préfixe viable* γ.

Comment ça marche ?

magie noire ...

Outils

Évidemment, tout ceci peut (doit !) être automatisé
outil classique : générateur d'analyseur LALR(1)

- yacc et bison en C/C++
- cup en Java
- PLY en Python
- GPPG en C#
- Yacc en Go
- Racc en Ruby
- ocamlyacc en OCaml
- Happy en Haskell
- Yecc en Erlang
- *insert your favorite language here*

Postparation

1. Terminez la première analyse par décalage réduction (transp. 47).
2. Dessinez l'arbre de dérivation correspondant.
3. Terminez l'automate LR(0) (solution sur chamilo, trouver l'erreur !)
4. Terminez la construction de la table SLR(1) (solution transp. 54, trouver l'erreur !)
5. Quel symbole représente chaque état de l'analyseur (transp. 54) ?

Bilan

Compilation (#4) : Grammaires attribuées

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

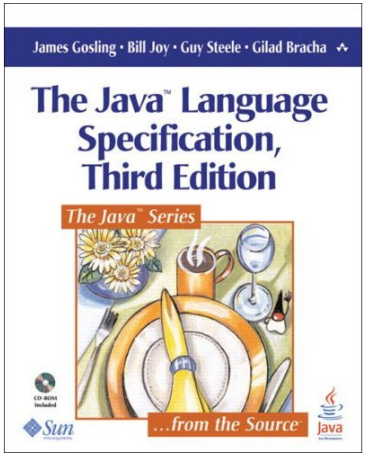
2022-2023

Meaning

How to define the meaning of programs in a given language ?

- Informal description most of the time (natural language, ISO, reference book. . .)
- Unprecise, ambiguous.

Informal Semantics



The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

It is recommended that code not rely crucially on this specification.

https://docs.oracle.com/javase/specs/jls/se10/html/jls-15.html#jls-15.7

Formal semantics

The formal semantics mathematically characterises the computations done by a given program:

- useful to design tools (compilers, interpreters).
- mandatory to reason about programs and properties of the language.

Ce que l'on veut

- Ajouter à la grammaire de l'information qui sera traitée pendant l'analyse syntaxique.
- Jusqu'à présent on n'a construit que des accepteurs.
- Nous voulons effectuer des actions/collecter de l'information à chaque étape de l'analyse syntaxique.

Une définition dirigée par la syntaxe est une généralisation d'une grammaire non contextuelle.

- chaque symbole de la grammaire possède un ensemble d'attributs :
  - synthétisés (calculés à partir des valeurs des attributs des fils)
  - hérités (calculés à partir des valeurs des attributs du père et des frères)
- chaque production A $\rightarrow \alpha$ de la grammaire possède un ensemble de règles sémantiques de la forme :

$$b = f(c_1, c_2, ...c_k)$$

où

- f est une fonction, et b est
  - soit un attribut synthétisé de A
  - soit un attribut hérité d'un des symboles de α
- $c_1, c_2, ...c_k$ sont des attributs de symboles quelconques de la production

On dit que b dépend des attributs $c_1, c_2, ...c_k$.

$$b = f(c_1, c_2, ...c_k)$$

où

- f est une fonction, et b est
  - soit un attribut synthétisé de A
  - soit un attribut hérité d'un des symboles de α
- $c_1, c_2, ...c_k$ sont des attributs de symboles quelconques de la production

On dit que b dépend des attributs $c_1, c_2, ...c_k$.

Grammaire attribuée : définition dirigée par la syntaxe dans laquelle les fonctions des règles sémantiques ne peuvent pas avoir d'effet de bord.

Definition

From Wikipedia:

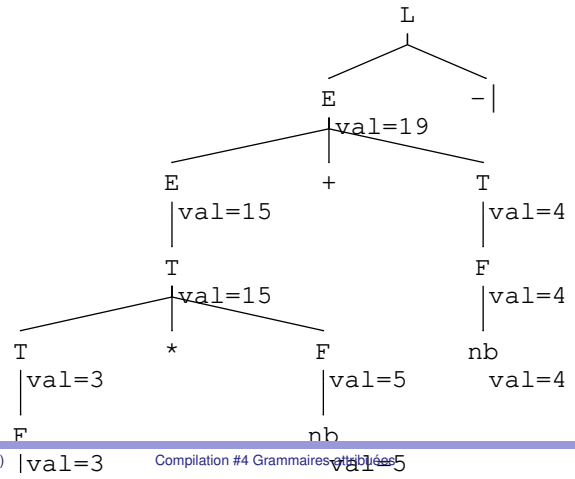
> *In computer science, an interpreter is a computer program that **directly executes instructions** written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.*

- An **interpreter** executes the input program according to the programming language **semantics**.

PRODUCTION	RÈGLE SÉMANTIQUE
$L \rightarrow E \dashv$	Imprimer(E.val)
$E \rightarrow E_1 + T$	E.val := E_1.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ **nb**	F.val := **nb**.val

Figure: Définition dirigée par la syntaxe d'une calculette

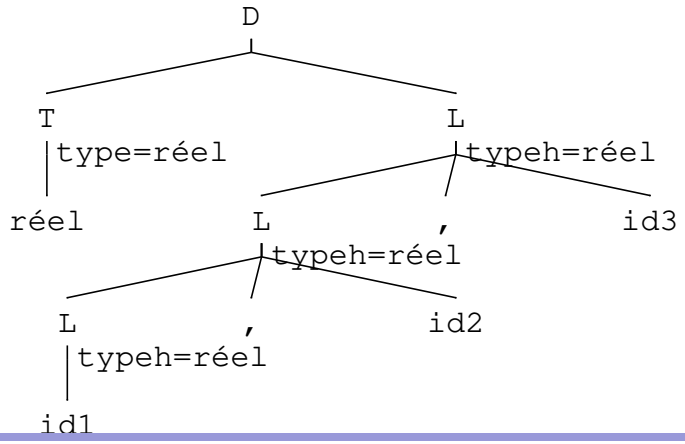
Arbre décoré pour la calculette



PRODUCTION	RÈGLE SÉMANTIQUE
D → T L	L.typeh := T.type
T → **entier**	T.type := entier
T → **réel**	T.type := réel
L → L_1 , **id**	L_1.typeh := L.typeh AjouterType(**id**.num, L.typeh)
L → **id**	AjouterType(**id**.num, L.typeh)

Figure: Définition dirigée par la syntaxe avec l'attribut hérité typeh

Arbre décoré pour la phrase `réel id1, id2, id3`



Graphe de dépendances

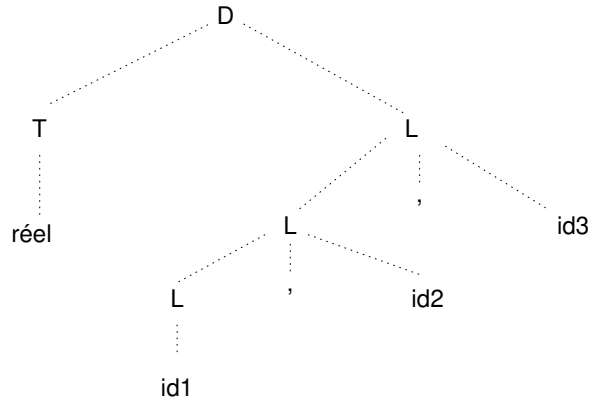
La valeur de certains attributs est calculée à partir de celle d'autres attributs, ce qui impose un ordre d'évaluation sur les règles sémantiques.
Les interdépendances entre attributs peuvent être décrites par un graphe orienté appelé graphe de dépendances. Ce graphe contient :

- un sommet pour chaque attribut,
- un arc de c à b si b dépend de c

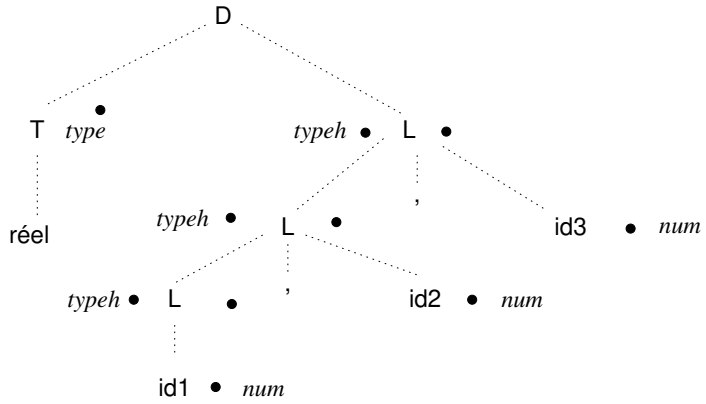
```
pour chaque nœud n de l'arbre syntaxique faire
   pour chaque attribut a du symbole de la
   grammaire étiquetant n faire
        construire un sommet dans le graphe de dép. pour a
pour chaque nœud n de l'arbre syntaxique faire
   pour chaque règle sémantique b:=f(c1, c2... ck)
   associée à la production appliquée en n faire
      pour i de 1 à k faire
           construire un arc du sommet correspondant à ci
           au sommet correspondant à b
```

Figure: Algorithme de construction du graphe de dépendances

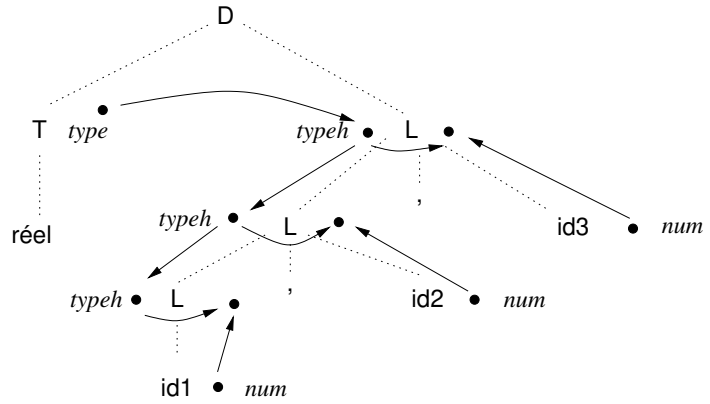
Graphe de dépendances pour l'arbre décoré précédent



Graphe de dépendances pour l'arbre décoré précédent



Graphe de dépendances pour l'arbre décoré précédent

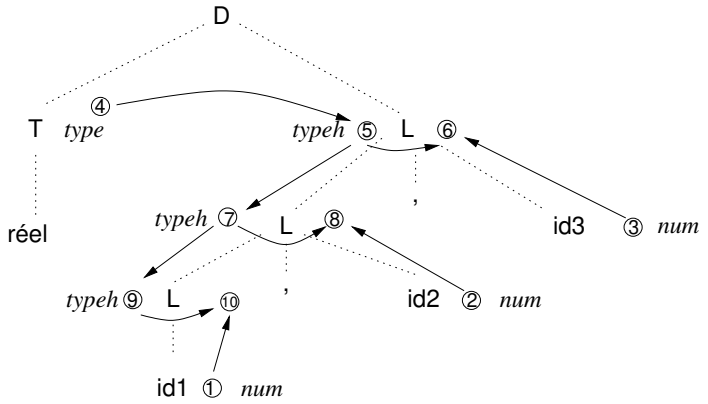


Tri topologique

Un tri topologique d'un graphe orienté acyclique est un ordonnancement quelconque $m_1, m_2 ... m_k$ des sommets du graphe tel que pour tous les arcs $m_i \rightarrow m_j$, m_i apparaît avant m_j dans l'ordonnancement.

Tout tri topologique d'un graphe de dépendances donne un ordre valide dans lequel les règles sémantiques peuvent être évaluées.

Graphe de dépendances pour l'arbre décoré précédent



- Une définition est S-attribuée si elle ne comporte que des attributs **s**ynthétisés.

- Une définition est S-attribuée si elle ne comporte que des attributs **s**ynthétisés.
- Une définition est L-attribuée si tout attribut hérité de X_j, $1 \leqslant j \leqslant n$ de la partie droite de la production A $\rightarrow$ X_1 X_2 ... X_n ne dépend que :
  1. des attributs des symboles X_1, X_2, ... X_{j-1} (à gauche de X_j dans la production)
  2. des attributs hérités de A.

- évaluation possible pendant l'analyse ascendante

- évaluation possible pendant l'analyse ascendante
- sans construire l'arbre
- l'analyseur conserve dans sa pile les symboles **et** leur attribut
- à chaque réduction, calcule l'attribut du symbole vers lequel il réduit

Cas des générateurs LALR
yacc/bison/cup/...

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	D

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	D
T_3*	nb_5+nb_4 ⊣	

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	D
T_3*	nb_5+nb_4 ⊣	D

Deleuze, Gonnord (Esisar) Compilation #4 Grammaires attribuées 2022-2023 ← 30 / 39 →

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	D
T_3*	nb_5+nb_4 ⊣	D
T_3*nb_5	+nb_4 ⊣	

Deleuze, Gonnord (Esisar) Compilation #4 Grammaires attribuées 2022-2023 ← 30 / 39 →

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	D
T_3*	nb_5+nb_4 ⊣	D
T_3*nb_5	+nb_4 ⊣	R7

Deleuze, Gonnord (Esisar) Compilation #4 Grammaires attribuées 2022-2023 ← 30 / 39 →

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	D
T_3*	nb_5+nb_4 ⊣	D
T_3*nb_5	+nb_4 ⊣	R7
T_3*F_5	+nb_4 ⊣	

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	D
T_3*	nb_5+nb_4 ⊣	D
T_3*nb_5	+nb_4 ⊣	R7
T_3*F_5	+nb_4 ⊣	R4

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	D
T_3*	nb_5+nb_4 ⊣	D
T_3*nb_5	+nb_4 ⊣	R7
T_3*F_5	+nb_4 ⊣	R4
T_{15}	+nb_4 ⊣	

Cas particuliers Définitions S-attribuées en ascendant

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	D
T_3*	nb_5+nb_4 ⊣	D
T_3*nb_5	+nb_4 ⊣	R7
T_3*F_5	+nb_4 ⊣	R4
T_{15}	+nb_4 ⊣	R3
...		

Exemple : la calculette sur 3*5+4⊣

Pile	Entrée	Action
	nb_3*nb_5+nb_4 ⊣	D
nb_3	*nb_5+nb_4 ⊣	R7
F_3	*nb_5+nb_4 ⊣	R5
T_3	*nb_5+nb_4 ⊣	D
T_3*	nb_5+nb_4 ⊣	D
T_3*nb_5	+nb_4 ⊣	R7
T_3*F_5	+nb_4 ⊣	R4
T_{15}	+nb_4 ⊣	R3
...		
E_{19} ⊣		R1 → affiche 19

Exemple bison

```
%%
toplevel : toplevel exp '\n' { printf("val is %d\n", $2); }
      | {}
;
exp : exp '+' term { $$ = $1 + $3;}
   | term { $$ = $1;}
;
term : term '*' factor { $$ = $1 * $3;}
    | factor { $$ = $1;}
;
factor : '(' exp ')' { $$ = $2;}
     | INTVAL { $$ = $1;}
     ;
%%
```


- analyseur à descente récursive
- chaque fonction retourne les attributs du nœud
  - appels récursifs donnent attributs des fils
  - calcul (règle sémantique de la production)

PRODUCTION	RÈGLE SÉMANTIQUE
$E \rightarrow T\ R$	E.val := T.val + R.val
$T \rightarrow P$	T.val := P.val
$T \rightarrow nb$	T.val := nb.val
$P \rightarrow (\ E\ )$	P.val := E.val
$R \rightarrow +\ E$	R.val := E.val
$R \rightarrow \varepsilon$	R.val := 0

Figure: Avec la grammaire du chapitre 2.a

Avec ANTLR

```
start : expr EOF {print($expr.v)} ;

expr returns [int v]
   : terme reste_expr {$v = $terme.v + $reste_expr.v } ;

terme returns [int v]
   : expr_par {$v = $expr_par.v }
   | NB {$v = $NB.int } ;

expr_par returns [int v]
   : '(' expr ')' {$v = $expr.v } ;

reste_expr returns [int v]
   : '+' expr {$v = $expr.v}
   | {$v = 0 } ;
```


Descente récursive

```
int parse(void) {
  int r = expression();
  consomme_ul(FDF);
  return r;
}

int expression(void) {
  int r1,r2;

  switch(prochaine_ul()) {
  case NB:
  case '(': r1=terme(); r2=reste_expression(); break;
  default: erreur();
  }
  return(r1+r2);
}

int terme(void) {
```

- en descendant : ok car on progresse de gauche à droite
  - appels récursif : attributs en paramètres
  - mais les hérités ne doivent pas dépendre de synthétisés...
- en ascendant : possible aussi à certaines conditions...

Bilan

Compilation (#5): AST and Typing.

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

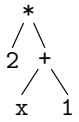
Objective

- Construction of our first intermediate representation for programs.
- Typing.

A bit about syntax

The texts: `2*(x+1)` and `(2 * ((x) + 1))` and `2 * /* comment */ ( x + 1 )`

have the same semantics ▶ they should have the **same internal representation**.



Example: syntax of expressions

The (abstract) grammar of arithmetic expressions is (avoiding parenthesis, syntactic sugar ...):

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{addition} \\
 & | & e \times e & \textit{multiplication} \\
 & | & \ldots &
\end{array}
$$

Remark : to properly define the semantics of the expression, it is sufficient to define $\mathcal{A}(e)$.

AST Definition (Wikipedia is your friend!)

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text.
The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes.

Warning

An AST is not! a derivation tree.

Give an example

Content

- Interpret with semantic actions : in the previous course.
- 2 other techniques here.

From grammar to an AST

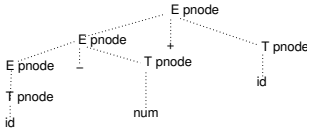
$E \rightarrow E + T \mid E - T$
$E \rightarrow T$
$T \rightarrow (E)$
$T \rightarrow \textbf{id} \mid \textbf{nb}$

```
node *create_node(top op, node *left, node *right)
node *create_leaf_id(int num)
node *create_leaf_nb(int value)
```

▶ **Attribution ?**

AST construction with attributes

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	E.pnode := create_node('+', E_1.pnode, T.pnode)
$E \rightarrow E_1 - T$	E.pnode := create_node('–', E_1.pnode, T.pnode)
$E \rightarrow T$	E.pnode := T.pnode
$T \rightarrow (E)$	T.pnode := E.pnode
$T \rightarrow$ **id**	T.pnode := create_leaf_id(**id**.num)
$T \rightarrow$ **nb**	T.pnode := create_leaf_nb(**nb**.val)



Example : expr to AST (in ANTLR w/Java backend)

AST Construction

```
prog returns [ AExpr e ] : expr EOF { $e=$expr.e; } ;

// We create an AExpr instead of computing a value
expr returns [ AExpr e ] :
| INT { $e=new AInt($INT.int); }
| LPAR x=expr RPAR { $e=$x.e; } // Parenthesis not represented in AST
| e1=expr PLUS e2=expr { $e=new APlus($e1.e,$e2.e); }
| e1=expr MINUS e2=expr { $e=new AMinus($e1.e,$e2.e); }
;
```

AST construction (main)

```
AExpr expr = parser.prog().e; // get the AST in var expr
// everything is ready to eval interpret (by tree traversal)
```


Example in Java - eval function

Evaluation is an eval function per class:

AExpr.java

```
public abstract class AExpr {
    abstract int eval(); // need to provide semantics
}
```

APlus.java

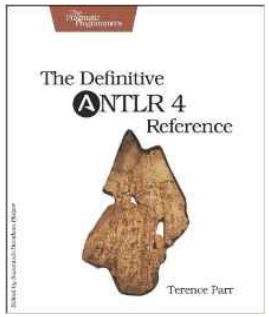
```
public class APlus extends AExpr {
    AExpr e1,e2;
    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }
    // semantics below
    int eval() { return (e1.eval()+e2.eval()); }
}
```

Principle - OO programming

The visitor design pattern is a way of separating an algorithm from an object structure on which it operates.[...] In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function.

https://en.wikipedia.org/wiki/Visitor_pattern

Application



Designing interpreters / tree traversal in ANTLR-Python

- The ANTLR compiler generates a Visitor class.
- We override this class to traverse the parsed instance.

Arit Example with ANTLR/Python 1/3

AritParser.g4

```
expr: expr mdop=(MULT | DIV) expr #multiplicationExpr
   | expr pmop=(PLUS | MINUS) expr #additiveExpr
   | atom #atomExpr
   ;

atom: INT #int
   | ID #id
   | '(' expr ')' #parens
   ;
```

- compilation with `-Dlanguage=Python3 -visitor`

Arit Example with ANTLR/Python 2/3 -generated file

AritVisitor.py (generated)

```
class AritVisitor(ParseTreeVisitor):
...
    # Visit a parse tree produced by AritParser#multiplicationExpr.
    def visitMultiplicationExpr(self, ctx):
        return self.visitChildren(ctx)


    # Visit a parse tree produced by AritParser#atomExpr.
    def visitAtomExpr(self, ctx):
        return self.visitChildren(ctx)
..
```

Arit Example with ANTLR/Python 3/3

Visitor class overriding to write the interpreter:

MyAritVisitor.py

```
class MyAritVisitor(AritVisitor):

    def visitInt(self, ctx):
        return int(ctx.getText())

    def visitMultiplicationExpr(self, ctx):
        leftval = self.visit(ctx.expr(0))
        rightval = self.visit(ctx.expr(1))
        if ctx.mdop.type == AritParser.MULT:
            return leftval * rightval
        else:
            return leftval / rightval
```

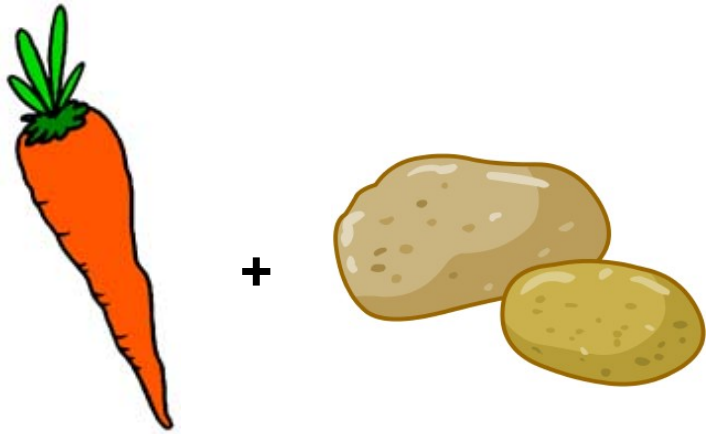

Arit Example with ANTLR/Python - Main

And now we have a full interpret for arithmetic expressions!

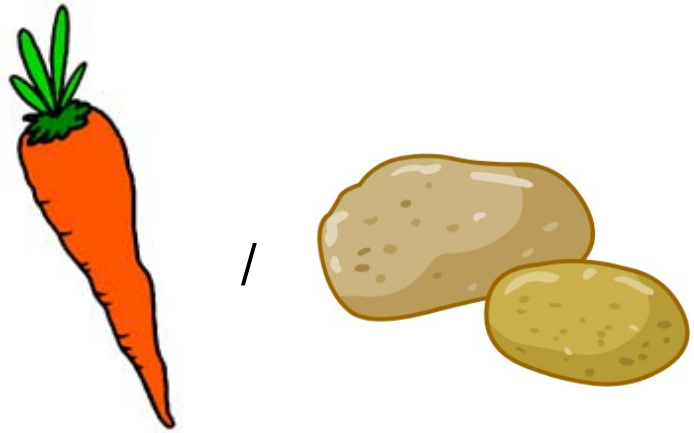
arit.py (Main)

```
lexer = AritLexer(InputStream(sys.stdin.read()))
stream = CommonTokenStream(lexer)
parser = AritParser(stream)
tree = parser.prog()
print("I'm here : nothing has been done")

visitor = MyAritVisitor()
visitor.visit(tree)
```


Typing



Typing



/

Typing

If you write: `"5432" + 1`
what do you want to obtain

- a compilation error? (OCaml)
- an exec error? (Python)
- the int 5433? (Visual Basic, PHP)
- the string `"54321"`? (Java)
- (too insane to put on a slide) (C, C++)
- anything else?

and what about `5432 / "1"` ?

Typing

When is

```
e1 + e2
```

legal, and what is its semantics?

- Typing: an analysis that gives a type to each subexpression, and reject incoherent programs.

When

- Dynamic typing (during execution): Lisp, PHP, Python
- Static typing (at compile time, after lexing+parsing): C, Java, OCaml

- Here: the second one.

Slogan

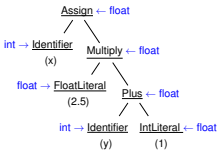
well typed programs do not go wrong

Typing objectives

- Should be **decidable**.
- It should reject programs like `(1 2)` in OCaml, or `1+"toto"` in C before an actual error in the evaluation of the expression: this is **safety**.
  The type system is related to the kind of error to be detected: **operations on basic types** / method invocation (message not understood) / correct synchronisation (e.g. session types) in concurrent programs / ...
- The type system should be expressive enough and not reject too many programs. (**expressivity**)

Principle

All sub-expressions of the program must be given a type



What does the programmer write?

- The type of all sub-expressions (like above) easy to verify, but tedious for the programmer
- Annotate only variable declarations (Pascal, C, Java, ...)
  ```
  {int x, y; x = 2.5 * (y + 1);}
  ```
- Only annotate function parameters (Scala)
  ```
  def toto(y : Int) { var x = 2.5 * (y + 1) }
  ```
- Annotate nothing: complete inference : Ocaml, Haskell, ...
  ```
  # let foo y = 2 * (y + 1);;
  val foo : int -> int = <fun>
  ```

Properties

- correction: “yes” implies the program is well typed.
- completeness: the converse.

(optional)

- principality : The most general type is computed.

What is a good output for a type-checker?

- We do not want:

  `failwith "typing error"`

  the origin of the problem should be clearly stated.
- We keep the types for next phases.

In practice

- Input: Trees are decorated by source code lines (and columns).
- Output: Trees are decorated by types in addition.

Mini-While (Abstract) Syntax

Expressions:

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{addition} \\
 & | & e \times e & \textit{multiplication} \\
 & | & \ldots &
\end{array}
$$

Mini-while:

$$
\begin{array}{rcll}
S(Smt) & ::= & x := expr & \text{assign} \\
 & | & skip & \text{do nothing} \\
 & | & S_1; S_2 & \text{sequence} \\
 & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\
 & | & \texttt{while } b \texttt{ do } S \texttt{ done} & \text{loop}
\end{array}
$$

Some vocabulary

We will define how to compute **typing judgements** denoted by:

$$\Gamma \vdash e : \tau$$

and means "in environment Γ, expression e has type τ" (here, $\tau \in \{int, bool\}$).

- Γ associates a type $\Gamma(x)$ to all declared variables x (that may or may not appear in e).

```
{
    int x = 42, y;
    float z;
}
```

$\rightsquigarrow$

$$\{ \quad x \rightarrow int, \quad y \rightarrow int, \quad z \rightarrow float$$

Typing rules format

We will use typing rules to prove these typing judgments (the fact that it is correct to do so is admitted). Typing rules are of the form $\frac{P_1 \dots P_n}{B}$ and mean $P_1 \wedge \dots P_n \Rightarrow B$.

Typing rules for expr

Here types are basic types: {`int`, `bool`}

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \texttt{int}} \quad (\text{or } \mathrm{tt}\colon \mathrm{bool}, \dots)$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

and plenty of similar rules.

Typing: an example

Considering $\Gamma = \{x_1 \mapsto int\}$, prove $\Gamma \vdash x1 + 9 : int$.

Typing rules for statements: $\Gamma \vdash S : \texttt{void}$

A statement S is well-typed in environment Γ, written: $\Gamma \vdash S : \texttt{void}$

on board!

- Example with expressions
- Guess/construct typing rules for assignment

Typing Mini-While: recap

$$\frac{c \in \mathbb{Z}}{\Gamma \vdash c : \texttt{int}} \qquad \frac{\Gamma(x) = t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash S_1 : \texttt{void} \quad \Gamma \vdash S_2 : \texttt{void}}{\Gamma \vdash S_1; S_2 : \texttt{void}} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x = e : \texttt{void}}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S_1 : \texttt{void} \quad \Gamma \vdash S_2 : \texttt{void}}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \texttt{void}}$$

Typing: an example

Considering $\Gamma = \{x_1 \mapsto int\}$, prove that the given sequence of instructions is well typed (with a correct proof tree using proof rules).

```
x1 = 3 ;
x1 = x1+9 ;
```

on board!

You can have a look at my YT video : https://youtu.be/2A-hQy_6YIE?t=808 at 13min30

Hybrid expressions

What if we have `1.2 + 42` ?

- reject?
- compute a float?

Hybrid expressions

What if we have `1.2 + 42` ?

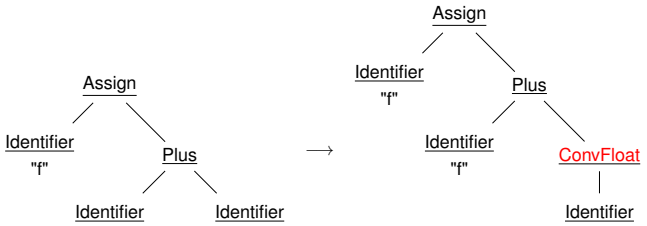
- reject? OCaml
- compute a float? Most mainstsream languages, e.g. C, Python, Java, etc.

Hybrid expressions

What if we have `1.2 + 42` ?

- reject? OCaml
- compute a float? Most mainstsream languages, e.g. C, Python, Java, etc.

▶ This is **type coercion**. Clean way to implement it: modify the AST to make the conversion explicit:



More complex expressions

What if we have types `pointer of bool`, or `array of int`? We might want to check equivalence (for addition ...).

▶ This is called **structural equivalence** (see Dragon Book, “type equivalence”). This is solved by a basic graph traversal checking that each element are equivalent/compatible.

Principle

- Γ is constructed going through the declaration part of the AST (may raise typing errors in case of double declarations for example)
- Γ is used to typecheck the program itself (e.g complain about type mismatch in an assignment or an expression, ...)
- One really wants typechecking on the AST, not the concrete program

Type Checking with a Visitor

MiniCTypingVisitor.py, rule for atoms

```
# In MiniC.g4 :
# expr : atom #atomExpr
#
# atom: OPAR expr CPAR #parExpr
# | INT #intAtom
# ...
def visitAtomExpr(self, ctx):
   return self.visit(ctx.atom())
```


Type Checking with a Visitor

MiniCTypingVisitor.py, rule for ”or”

```
# In MiniC.g4 :
# expr: expr OR expr #orExpr
def visitOrExpr(self, ctx):
   ltype = self.visit(ctx.expr(0))
   rtype = self.visit(ctx.expr(1))
   if BaseType.Boolean == ltype and BaseType.Boolean == rtype:
      return BaseType.Boolean
   else:
      self._raise(ctx, 'boolean operands', ltype, rtype)
```


In practice for MiniC (lab sessions)

- No annotation is added to the AST (MiniC is simple enough, they are not needed)
- Typing is given (may serve as an example visitor ...)

Summary

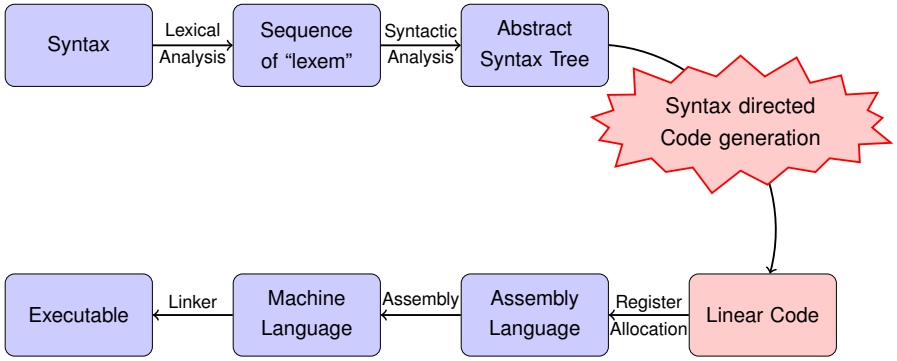
Compilation (#6): Syntax-Directed Code Generation

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

Big Picture



Nice picture from G. Radanne

Rules of the Game here

For this code generation:

- Still no functions and no non-basic types. (MiniC w/o functions and strings)
- Syntax-directed: one grammar rule → a set of instructions. ▶ Code redundancy.
- No register reuse: everything will be stored on the stack.

The Target Machine: RISCV.

Code Generation vs Memory/Register Allocation

- Code generation in two steps:
  1. Generate instructions without deciding where data is stored (put everything in temporaries)
  2. Decide where each temporary is allocated (register? stack?)
- Temporary (sometimes called "virtual register"): temporary where data can be stored. Difference with (physical) registers:
  - They don't exist in the real processor / instruction set
  - There are an infinity of them

A first example (1/2)

How do we translate:

```
int x, y;
x=4;
y=12+x;
```

- Variable decl's visitor gives a temporary to each variable: $x \mapsto temp0$, $y \mapsto temp1$.
- Compute 4, store somewhere, then copy in x's temporary.
- Compute $12 + x$: 12 in `temp2`, copy the value of x in `temp3`, then add, store in `temp4`, then copy into y (i.e. `temp1`).

▶ Create temporaries whenever needed.

A first example: 3@code (2/2)

"Compute 4 and store in x (temp0)":

```
li temp2, 4
mv temp0, temp2
```

Give the expected code for `y=12+x`?

Objective

3-address RISCV **Code Generation** for the Mini-While language:

- All variables are int/bool.
- All variables are global.
- No functions

with syntax-directed translation. Implementation in Lab (MiniC)

▶ This is called **three-adress code generation**

Abstract Syntax

Expressions:

e	::=	c	constant
	|	x	variable
	|	$e + e$	addition
	|	e or e	boolean or
	|	$e < e$	less than
	|	...	

Statements:

S	::=	$x := expr$	assign
	|	$skip$	do nothing
	|	$S_1; S_2$	sequence
	|	`if` b `then` S_1 `else` S_2	test
	|	`while` b `do` S `done`	loop

Code generation utility functions

We will use:

- A new (fresh) temporary can be created with a `fresh_tmp()` function.
- A new fresh label can be created with a `new_label()` function.
- The generated instructions are close to the RISCV ones.

Code generation for expressions, example

e ::= c (cst expr)	

	`dr <- fresh_tmp()`
	`code.add(li(dr, c))`
	`return dr`

▶ this rule gives a way to generate code for any constant. **Do it for constant 4!**

Code generation for a boolean expression, example

e ::= $e_1 < e_2$	

```
dr <-fresh_tmp()
t1 <- GenCodeExpr(e1)
t2 <- GenCodeExpr(e2)
endrel <- new_label()
code.add(li(dr, 0))
#if t1>=t2 jump to endrel
code.add("bge endrel,  t1, t2")
code.add(li(dr, 1))
code.addLabel(endrel)
return dr
```

- the destination register contains after execution an integer value 0 or 1. **Try (e::=x<4)**

Second example: a boolean test

Let us generate the code for $x < 4$ (assuming x is stored in `temp0`):

```
li temp3, 4 // get 4
li temp2, 0
geq temp0, temp3, lbl0 // >= comp + jump
li temp2, 1
lbl0:
```


Code generation for commands, example

if b then $S1$ else $S2$	

```
lelse,lendif <- new_labels()
t1 <- GenCodeExpr(b)
#if the condition is false, jump to else
code.add("beq lelse, t1,  0")
GenCodeSmt(S1) #then
code.add(j(lendif))
code.addLabel(lelse)
GenCodeSmt(S2) #else
code.addLabel(lendif)
```

Do it for `if x<4 then y=7 else ...` - next slide

Example for tests.

Let us generate the code for `if (x<4) then y=7 else ...` (y in `temp1`)

```
## code from previous slide here to compute x<4
beq temp2, zero, lelse1 // if false, jump
li temp4, 7
mv temp1, temp4 // y gets 7
j lendif1 // don't forget this one!
lelse1:
  # code for else branch
lendif1:
```

From 3@ code to valid RISCV

3@code is not valid RISCV code!

We explore 3 allocation algorithms:

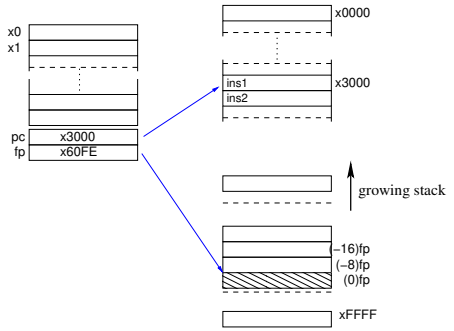
- All in registers $temp_i \rightarrow register$ ← very, very naive
- All in memory $temp_i \rightarrow memory$ ← very naive
- Something in between ← yes, we'll do smart stuff too :-)

A stack, why?

- Store constants, strings, ...
- Provide an easy way to communicate arguments values (see later)
- Give place to store intermediate values (e.g. 2∗3 in `x = 1 + 2 * 3`)

Stack with RISCV

- There is a special register `fp`.
- Store and loads from `fp`



Nice picture by N. Louvet - adapted in 2019

How to store into the stack

Store (the content of) s_3 on the stack at offset `offset`:

```
sd s3, -offset*8(fp)
# To generate from Python:
# sd(s3, Offset(FP, -offset*8))
# "write the value of s3 at address fp - offset*8"
```


Code Generation

Input: a MiniC file:

```
int main(){
int n;
n=6;
return 0;}
```

Output: a RISCV file:

```
1  [...]
2        ;; (stat (assignment n = (expr (atom 6)) ;))
3        li t1, 6    ; t1 is a riscv register.
4        mv t2, t1
5  [...]
```


Steps of lab 4

- 3-address code generation according to the code generation rules.
- Simple register/memory allocation (naive, all-in-mem) + pretty print.

Drawbacks of the former translation

Drawbacks:

- redundancies (constants recomputations, . . .)
- memory intensive loads and stores.

▶ we need a more efficient data structure to reason on: **the control flow graph (CFG)**. (see next course)

Summary : 3adress code generation

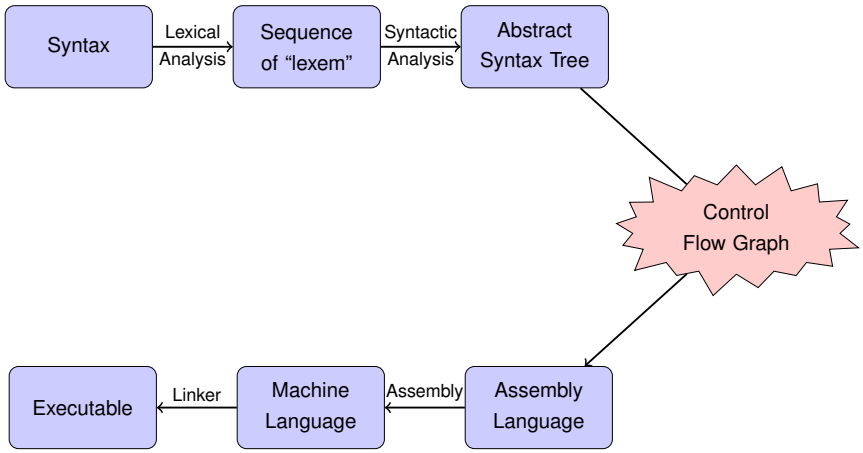
Compilation (#7) : Intermediate Representations: CFG, DAGs, and local optimisations (Instruction Selection and Scheduling)

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

Big Picture



3 address construction "problems"

Temporary reuse ?

```
li temp3, 4
mv temp0, temp3
;; temp3 is never used again
li temp4, 0
mv temp1, temp4
```

`temp3` and `temp4` could be mapped to the same physical location.

```
li temp5, 4
bge ..., foo
;; temp5 not used.
;; Its physical location
;; can be shared.
j end
foo:
;; temp5 used
end
```


A first IR

We thus need a better data structure to propagate and infer information. We need:

- A data structure that helps us to reason about the flow of the program.
- Which embeds our three address code.

▶ Control-Flow Graph.

Definitions

Definition (Basic Block)

Basic block: largest (3-address RISCXX*) instruction sequence without label. (except at the first instruction) and without jumps and calls.*

Definition (CFG)

It is a directed graph whose vertices are basic blocks, and edge $B_1 \to B_2$ exists if B_2 can follow immediately B_1 in an execution.

- two optimisation levels: local (BB) and global (CFG)

An example 1/2

Let us consider the program:

```
int x,y;
if (x<4) y=7; else y=42;
x=10;
```

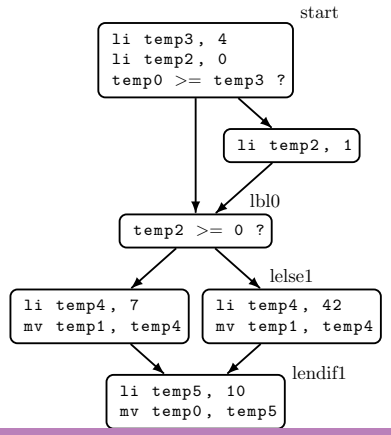
We already generated the (linear code) for a large part of it.

An example 2/2

```
  li temp3, 4
  li temp2, 0
  bge temp0, temp3, lbl0
  li temp2, 1
lbl0:  # if false, jump (skip the 'then')
  bge temp2, 0, lelse1
  li temp4, 7
  mv temp1, temp4  # y gets 7
  jump lendif1
lelse1:
  li temp4 42
  mv temp1, temp4  # y gets 42
lendif1:
  li temp5, 10
  mv temp0, temp5  # end
```

An example 2/2

```
  li temp3, 4
  li temp2, 0
  bge temp0, temp3, lbl0
  li temp2, 1
lbl0:  # if false, jump (skip the 'then')
  bge temp2, 0, lelse1
  li temp4, 7
  mv temp1, temp4  # y gets 7
  jump lendif1
lelse1:
  li temp4 42
  mv temp1, temp4  # y gets 42
lendif1:
  li temp5, 10
  mv temp0, temp5  # end
```



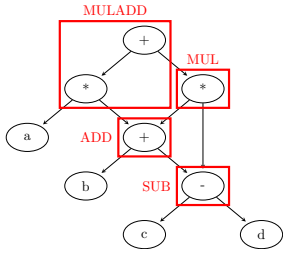
Identifying Basic Blocks (from 3 address code)

- The first instruction of a basic block is called a **leader**.
- We can identify leaders via these three properties:
  1. The first instruction in the intermediate code is a leader.
  2. Any instruction that is the target of a conditional or unconditional jump is a leader.
  3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
- Once we have found the leaders, it is straighforward to find the basic blocks: for each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.

Big picture (Basic Block Optimisation)

- Front-end → a CFG where nodes are basic blocks.
- Basic blocks → DAGs that explicit common computations

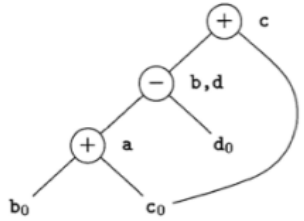
```
u1 := c - d
u2 := b + u1
u3 := a * u2
u4 := u2 * u1
u5 := u3 + u4
```



▶ choose instructions(**selection**) and order them (**scheduling**).

An Example of BB DAG construction

```
a = b + c
b = a - d
c = b + c
d = a - d
```



Useful links : https://www.youtube.com/watch?v=PXTKWvyQUwE and

https://www.cse.iitm.ac.in/~krishna/cs3300/pm-lecture3.pdf for other BB optimisations.

Instruction Selection, in general

The problem:

- a list of instructions/operations that compute one or more expressions.
- map these operations in “real machine instructions”.
- at minimum cost.

Instruction Selection

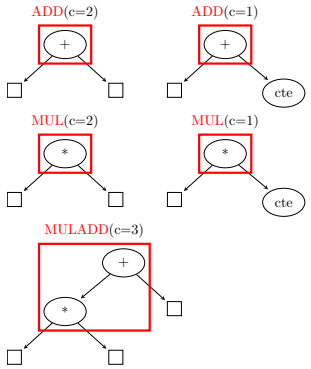
The problem of selecting instructions is a DAG-partitioning problem. But what is the objective ?

The best instructions:

- cover bigger parts of computation.
- cause few memory accesses.

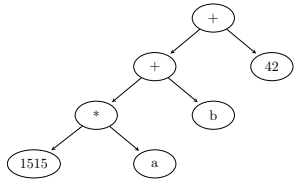
▶ Assign a cost to each instruction, depending on their addressing mode.

Instruction Selection: an example



(Our RISCXX has no MULADD instruction nor "add with constants", this is just an example).

What is the optimal instruction selection for:



▶ Finding a tiling of minimal cost: it is **NP-complete** (SAT reduction).

Tiling trees / DAGs, in practice

For tiling:

- There is an optimal algorithm for **trees** based on dynamic programming.
- For DAGs we use heuristics (decomposition into a forest of trees, . . .)

▶ The literature is plethoric on the subject.

Instruction Scheduling, in general

The problem:

- change the order of instructions.
- to “optimise’.
- without “cutting dependencies”.

Instruction Scheduling, what for?

We want an evaluation order for the instructions that we choose with **Instruction Scheduling**.

A scheduling is a function θ that associates a **logical date** to each instruction. To be correct, it must respect data dependencies:

```
(S1) u1 := c - d
(S2) u2 := b + u1
```

implies $\theta(S_1) < \theta(S_2)$. We can choose $\theta(S_1) = 0, \theta(S_2) = 1$

▶ How to choose among many correct schedulings? depends on the target architecture.

Architecture-dependant choices

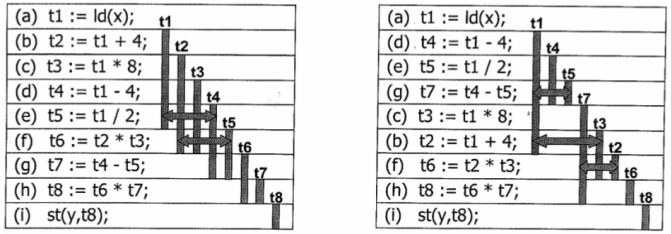
The idea is to exploit the different ressources of the machine at their best:

- instruction parallelism: some machines have parallel units (subinstructions of a given instruction).
- prefetch: some machines have non-blocking load/stores, we can run some instructions between a load and its use (hide latency!)
- pipeline.
- registers: see next slide.

(sometimes these criteria are incompatible)

Register use

Some schedules induce less **register pressure**:



In this picture the dates of the instructions are implicit : line 1 is date 1, line 2 is date 2. . .

▶ How to find a schedule with less register pressure?

Scheduling wrt register pressure

Result: this is a linear problem on trees, but NP-complete on DAGs (Sethi, 1975).

- Sethi-Ullman algorithm on trees, heuristics on DAGs

A slight variation of this algorithm can be found on Wikipedia, the leaves values here are chosen equal to 1 since our machine does not have any direct access to constant values.

Sethi-Ullman algorithm on trees

$\rho(node)$ denoting the number of (pseudo)-registers necessary to compute a node:

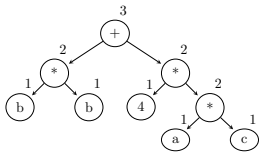
- $\rho(leaf) = 1$
- $\rho(nodeop(e_1, e_2)) = \begin{cases} max\{\rho(e_1), \rho(e_2)\} & \text{if } \rho(e_1) \neq \rho(e_2) \\ \rho(e_1) + 1 & \text{else} \end{cases}$

(the idea for non "balanced" subtrees is to execute the one with the biggest ρ first, then the other branch, then the op. If the tree is balanced, then we need an extra register)

- then the code is produced with postfix tree traversal, the biggest register consumers first.

Sethi-Ullman algorithm on trees - an example

Min number of (additional) registers for $b^2 + 4ac$ with a,b,c already in registers ?



The tree traversal then produces the following code:

	tmp_1	tmp_2	tmp_3	tmp_4
`mul tmp1, b, b`				
`mul tmp2, a, c`	■			
`li tmp3, 4`	■	■		
`mul tmp4, tmp2, tmp3`	■	■	■	
`add tmp5, tmp1, temp4`	■		■	■

cells in black denote for each instruction the set of entry alive temporaries.

Conclusion (instruction selection/scheduling)

Plenty of other algorithms in the literature:

- Scheduling DAGs with heuristics, ...
- Scheduling loops **in advanced compilation courses**

Bilan

1. Control flow Graph

2. Local optimizations
   - Basic Blocks DAG Construction
   - Instruction Selection
   - Instruction Scheduling

Compilation (#8):

Register Allocation + Data Flow Analyses

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

Where are we ?

source code

↓

lexical+syntactic analysis + typing

↓

decorated AST

↓

code production (numerous phases)

↓

assembly language

- We work on IRs (Middle-end).

Credits

Fernando Pereira's course on register allocation:

http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/RegisterAllocation.pdf

What for ?

- Finding storage locations to the values manipulated by the program ▶ registers or memory.
- registers are fast but in small quantity.
- memory is plenty, but slower access time.

▶ A good register allocator should strive to keep in registers the variables used more often.

> "Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important - if not the most important - of the optimizations."

Hennessy and Patterson (2006) - [Appendix B; p. 26]

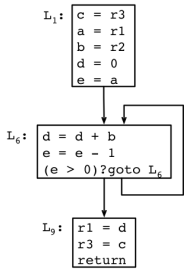
What for?

Expected behavior of **register allocation**:

- Input: a CFG with basic blocks with 3-address code (and pseudo-registers, aka temporaries)
- Output: same CFG but without temporaries:
  - replace with physical registers as much as possible.
  - if not **spill**, ie allocate a place in memory.
  - all copies assigned to the same physical registers ("moves") can be removed: **coalescing** (**optional**).

Register constraints

Some variables are assigned to some specific registers (compiler, architecture constraints)



▶ r1,r2,r3 are used to pass function arguments here (thus should be reserved)

The key notion: liveness

Observation

Two variables that are simultaneously **alive** must be assigned different registers.

(formal definition of alive follows)

Register assignment is NP-complete

Problem to solve

Given P a program and K general purpose registers, is there an assignment of the variables P in registers, such that (i) every variable gets at least one register along its entire live range, and (ii) simultaneously live variables are given different registers ?

Gregory Chaitin has shown, in the early 80's, that the register assignment problem is NP-Complete (register allocation via coloring, 1981)

3-phase algorithm

- **Liveness analysis**
  - When is a given value necessary for the rest of the computation?
- **Interference graph**
  - A graph that encodes which temporaries cannot be mapped to the same location.
- **Graph coloring** then register allocation.
  - The effective allocation: physical registers and stack allocation for temporaries.

Liveness analysis

In the sequel we call **variable** a temporary or a physical register.

Definition (Alive Variable)

In a given program point, a variable is said to be alive if the value it contains may be used in the rest of the execution.

"May", and non-decidable property ⇒ overapproximation.

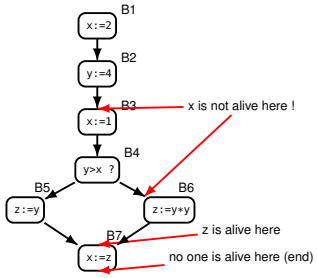
Important remark: here a block = a statement/program point. We have the same kind of analyses with block=basic block.

An example for live ranges

Definition

A variable is **live** at the exit of a block if there exists a path from the block to a use of the variable that does not redefine the variable.

```
x:=2;
y:=4;
x:=1;
if (y>x)
   then z:=y
   else z:=y*y ;
x:=z;
```



- The information flow is **backward**: from uses to definitions.

Data flow expressions

Definition

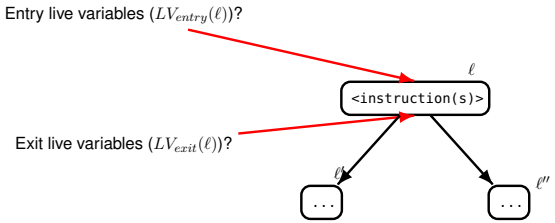
A variable that appears on the left hand side of an assignment is **killed** by the block. Tests do no kill variables.

Definition

A **generated** variable is a variable that appears in the block (read-only!)

- We precompute the sets : $kill_{LV}(block)$ and $gen_{LV}(block)$ for all blocks.

Dataflow equations



$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell \text{ is final} \\ \bigcup_{\ell' \in succ_G(\ell)} LV_{entry}(\ell') & \text{else} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

Dataflow equations: solving

Important Remark

Fixpoint equations $X = F(X)$!

Here:

- Initialise LV sets to $\emptyset$.
- Compute LV_{entry} sets, then LV_{exit}, and continue.
- Stop when a fix point is reached.

- (vector of) Sets are strictly growing, and the live range set is at most the set of all variables, thus **this algorithm terminates**.

Steps

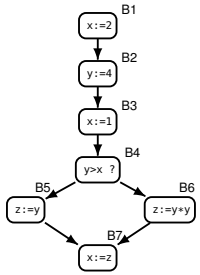
$In(\ell)$, $Out(\ell)$ are temporary sets of variables. After convergence, they will be equal to $LV_{entry}(\ell)$, $LV_{exit}(\ell)$.

			Step 1		Step 2		Step 3 (stable)
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$
1	$\{x\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{y\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{y\}$	$\emptyset$
3	$\{x\}$	$\emptyset$	$\emptyset$	$\{x,y\}$	$\{y\}$	$\{x,y\}$	$\{y\}$
4	$\emptyset$	$\{x,y\}$	$\{x,y\}$	$\{y\}$	$\{x,y\}$	$\{y\}$	$\{x,y\}$
5	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
6	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
7	$\{x\}$	$\{z\}$	$\{z\}$	$\emptyset$	$\{z\}$	$\emptyset$	$\{z\}$

init to empty sets is implicit in this slide.

Final result and use

Backward analysis and we want the smallest sets, here is the final result : (we assume all vars are dead at the end).



ℓ	$LV_{entry}(\ell)$	$LV_{exit}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{y\}$
3	$\{y\}$	$\{x,y\}$
4	$\{x,y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	$\emptyset$

- Use : Dead code elimination !

Step 2: Interferences

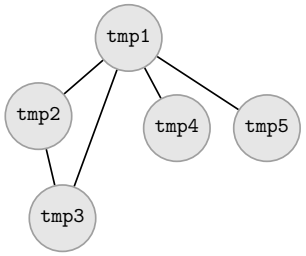
Here is the output of the liveness analysis for $a + (b + c)$:

	tmp_1	tmp_2	tmp_3	tmp_4	tmp_5	tmp_6
`load tmp1,la`						
`load tmp2,lb`	■					
`load tmp3,lc`	■	■				
`ADD tmp4, tmp2, tmp3`	■	■	■			
`MV tmp5, tmp4`	■		■	■		
`ADD tmp6, tmp1, tmp5`	■			■	■	
⋮						■

- `tmp1` is in conflict with `tmp2` (because of instruction 3) denoted by $tmp_1 \bowtie tmp_2$.

Interference graph

The relation $\bowtie$ defines a conflict/interference graph:



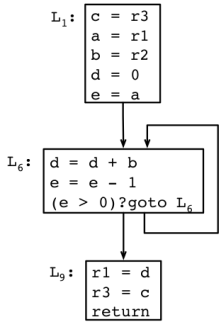
We want a **correct allocation** with respect to $\bowtie$:

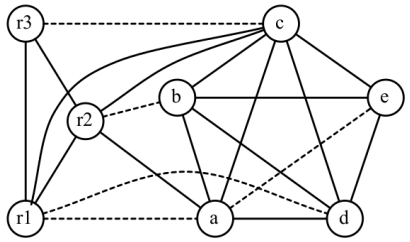
$tmp_1 \bowtie tmp_2 \implies Alloc(tmp_1) \neq Alloc(tmp_2)$.

- Graph coloring.

Running example

Important: in this example consider the r_i as temporary registers, like the others.

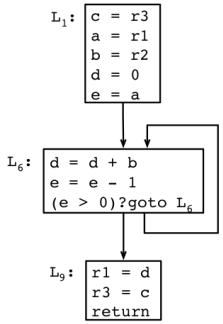


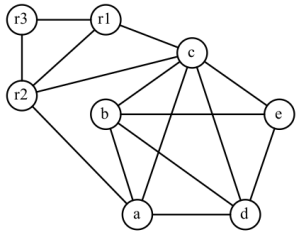


Dashed edges represent moves!

Running example

Important: in this example consider the r_i as temporary registers, like the others.





Let's look at the graph without moves first

Kempe's simplification algorithm 1/2

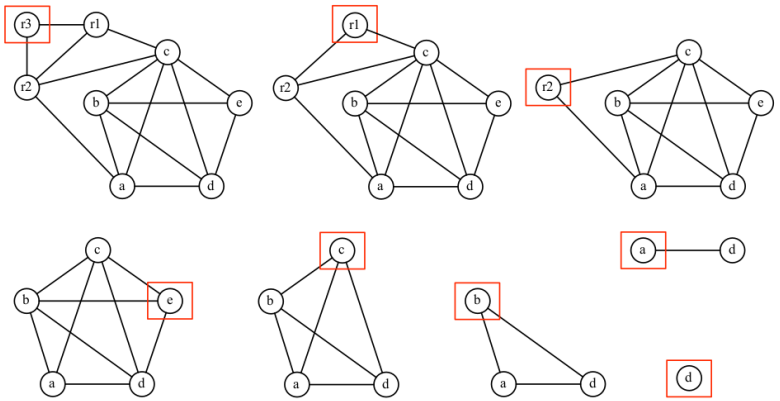
On the interference graph (without coalesce edges):

Proposition (Kempe 1879)

Suppose the graph contains a node m with fewer than K neighbours. Then if $G' = G \setminus \{m\}$ can be K-colored, then G can be K-colored as well.

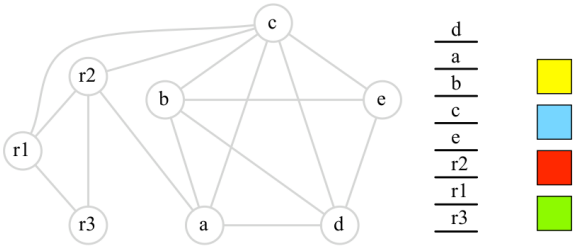
- Pick a low degree node, and remove it, and continue until remove all (the graph is K-colorable) or ...

Kempe's simplification algorithm 2/2

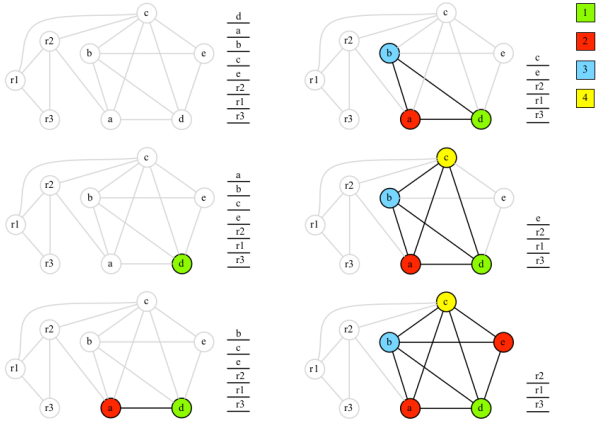


Let's color! ("Kempe's heuristic")

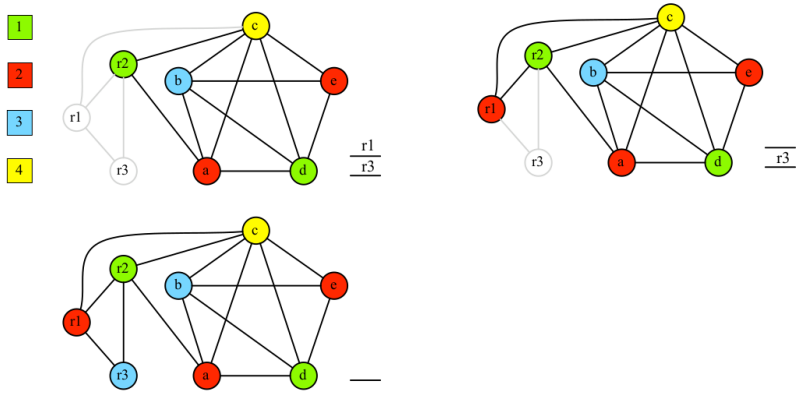
- We assign colors to the nodes greedily, in the reverse order in which nodes are removed from the graph.
- The color of the next node is the first color that is available, i.e. not used by any neighbour.



Greedy coloring example 1/2



Greedy coloring example 2/2



On the number of colors (K)

In the last example, we chose K=4, and this is nice, because the graph is 4-colorable.

- The given heuristic may fail to color the graph with K colors: it doesn't mean that the graph is not K-colorable (**heuristic**!).
- We can chose:
  - either to eliminate the "non-colorable node" of the graph and continue with the other nodes inside the node stack.
  - either to augment the K parameter.

Recall memories - Final code generation

With a 3 address code + allocation, rewrite each 3 address instruction into "real code":

- Each temporary is rewritten into his allocated physical register.
- If the temporary is in memory (Spilling), we generate code with appropriate loads and stores.

If the graph was not successfully colored

Non-colored variables[1] are named **spilled temporaries**.
There are many solutions to handle spilled variables.

[1] either not colored at all or colored with number >K

A naive solution: also color memory!

- Launch the coloration algorithm with an infinite number of colors:
  - first colors are mapped to registers (used in priority by the coloring algorithm)
  - other colors are mapped to offsets in the stack, i.e. spilled to memory
- Drawback: we need a few registers to implement the spilling

More sophisticated: Live range splitting

Idea: Modify the code to lower the number of simultaneously alive registers.
Invent 2 versions of the same variable (**live-range splitting**), and modify the code into:

```
ADD temp51, temp4, temp3
STORE temp51, [locationinmemory] # replace with actual location
..
LOAD temp52, [locationinmemory]  #same
ADD temp6, temp52, #5
```

▶ But now we have to allocate these two new variables!

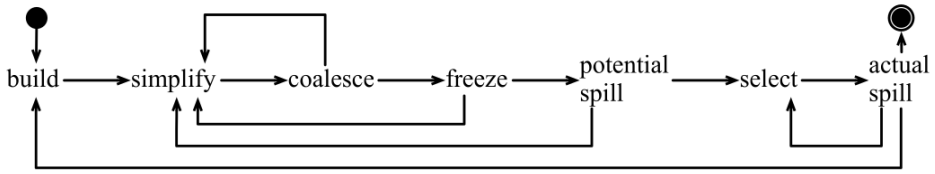
We relaunch the coloring algorithm. This is called **Iterative Register Allocation**.

To go further: Iterative Register Coalescing[2]

Two new optimizations to improve register allocation further

1. Register coalescing
2. Clever spilling

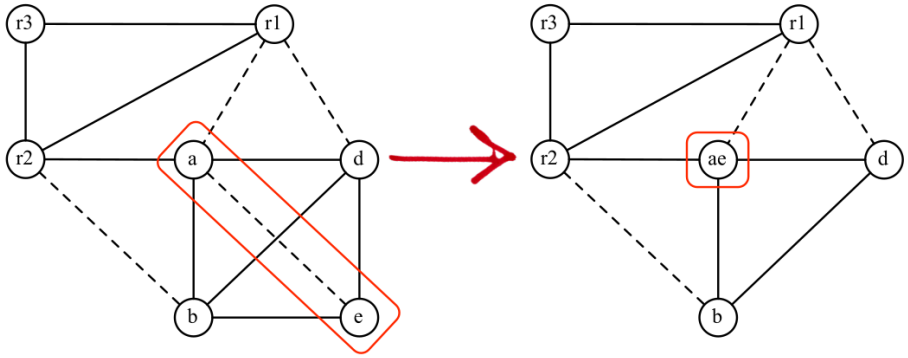
An iterative algorithm with many steps:



[2] Iterated Register Coalescing, TOPLAS (1996)

Iterative Register Coalescing – Coalescing

Coalescing consists of collapsing two move related nodes together

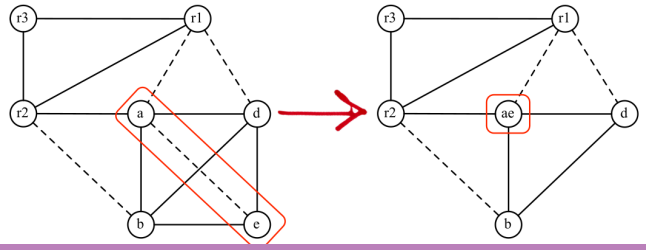


Iterative Register Coalescing – Coalescing

Two heuristics for coalescing safely:

Briggs Nodes a and b can be coalesced if the resulting node ab will have fewer than K neighbors of high degree (i.e., $degree \geqslant K$ edges)

George Nodes a and b can be coalesced if, for every neighbor t of a, either t already interferes with b, or t is of low degree.



Iterative Register Coalescing – Spilling

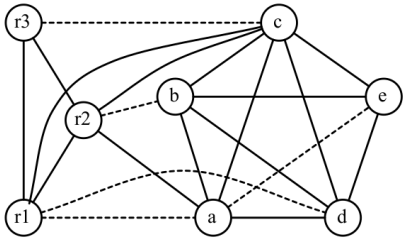
- How to choose which variables to spill ? This is actually really hard:
  - We want to spill variables that are less used dynamically
  - We only have static information

We can use a heuristic:

```
SPILLCOST(v)
  cost = 0
  foreach definition or use in block B
    cost += 10N/D, where
      N is B's loop nesting factor
      D is v's degree in the interference graph
```

Other Algorithms

- **Linear scan**: greedy coloring of interval graphs. (see Fernando Pereira's slides on register allocation: 18 to 35)
- Plenty of other heuristics for spilling.



Bilan

1. Register allocation - Intro
2. Live-range Information: Liveness Analysis
3. Register Allocation with graph coloring
   - Conflict (Interference) Graph
   - Coloring
   - Spilling strategies

Compilation (#9):

Dataflow analyses for code optimisation and safety

C. Deleuze & L. Gonnord

Grenoble INP/Esisar

2022-2023

Slides of this introduction

Credits : F. Pereira, UFMG

Goal 1: Program Optimization

- One of the goals of the techniques that we will see in this course is to optimize programs.
- There are many, really many, different ways to optimize programs. We will see some of these techniques:
  - Copy elimination
  - Constant propagation
  - Lazy Code Motion
  - Register Allocation
  - Loop Unrolling
  - Value Numbering
  - Strength Reduction
  - Etc, etc, etc.

```
#include <stdio.h>
#define CUBE(x) (x)*(x)*(x)
int main() {
  int i = 0;
  int x = 2;
  int sum = 0;
  while (i++ < 100) {
    sum += CUBE(x);
  }
  printf("%d\n", sum);
}
```

How can you optimize this program?

```
_main:
  pushl %ebp
  movl  %esp, %ebp
  pushl %ebx
  subl  $20, %esp
  call  L3
L3:
  popl  %ebx
  movl  $800, 4(%esp)
  movl  %eax, (%esp)
  call  _printf
  addl  $20, %esp
  popl  %ebx
  leave
  ret
```

Goal 2: Bug Finding

- Compiler analyses are very useful to find (and sometimes fix) bugs in programs.
  - Null pointer dereference
  - Array out-of-bounds access
  - Invalid Class Cast
  - Tainted Flow Vulnerabilities
  - Integer Overflows
  - Information leaks

```
1 void read_matrix(int* data,
      char w, char h) {
2   char buf_size = w * h;
3   if (buf_size < BUF_SIZE) {
4     int c0, c1;
5     int buf[BUF_SIZE];
6     for (c0 = 0; c0 < h; c0++) {
7       for (c1 = 0; c1 < w; c1++) {
8         int index = c0 * w + c1;
9         buf[index] = data[index];
10      }
11    }
12    process(buf);
13  }
14 }
```

Can you spot a security bug in this program. **Be aware**: the bug is tricky.

Static And Dynamic Analyses

- Compilers have two ways to understand programs:
  - Static Analysis
  - Dynamic Analysis
- Static analyses try to discover information about a program without running it.
- Dynamic analyses run the program, and collect information about the events that took place at runtime.

1) Can you give examples of dynamic analyses?

2) And can you give examples of static approaches?

3) What are the pros and cons of each approach?

Dynamic Analyses

- Dynamic analyses involve executing the program.
  - **Profiling**: we execute the program, and log the events that happened at runtime. Example: `gprof`.
  - **Test generation**: we try to generate tests that cover most of the program code, or that produce some event. Example: `Klee`.
  - **Emulation**: we execute the program in a virtual machine, that takes care of collecting and analyzing data. Example: `valgrind`, and `CFGGrind`.
  - **Instrumentation**: we augment the program with a meta-program, that monitors its behavior. Example: `AddressSanitizer`.

Static Analyses

- In this course we will focus on static analyses.
- There are three main families of static analyses that we will be using:
  - **Dataflow analyses**: we propagate information based on the dependences between program elements, which are given by the syntax of the program.
  - **Constraint-Based analyses**: we derive constraints from the program. Relations between these constraints are not determined explicitly by the program syntax.
  - **Type analyses**: we propagate information as type annotations. This information lets us prove properties about the program, such as progress and preservation.

Here

Dataflow analyses (like in the previous course)

60's

Early Code Optimizations

- Frances E. Allen, working alone or jointly with John Cocke, introduced many of the concepts for optimization:
  - Control flow graphs
  - Many dataflow analyses
  - A description of many different program optimizations
  - Interprocedural dataflow analyses
  - Worklist algorithms
- A lot of these inventions and discoveries have been made in the IBM labs.

70's

The Dataflow Monotone Framework

- Most of the compiler theory and technology in use today is based on the notion of the dataflow monotone framework.
  - Propagation of information
  - Iterative algorithms
  - Termination of fixed point computations
  - The meet over all paths solution to dataflow problems
- These ideas came, mostly, from the work of Gary Kildall, who is one of the fathers of the modern theory of code analysis and optimization.

The inventor of the BIOS system!

In addition of being paramount to the development of modern compiler theory, Gary Kildall used to host a talk show called "*The Computer Chronicles*". Nevertheless, he is mostly known for the deal with the Ms-DOS system that involved IBM and Bill Gates.

Recall memories: liveness

In the previous course, we saw how to propagate liveness information:

- as set of live variables
- from bottom to top
- with union to merge information.

Common subexpressions / Available expressions

Avoiding the computation of an (arithmetic) expression :

```
x:=a+b;
y:=a*b;
while(y>a+b) do
   a:=a+a;
   x:=a+b;
done
```


AE : genesis of the analysis

An expression is available at a control point if its current value has already been computed earlier in the execution:

- Sets of expressions.
- How does information originate? from a use that do not redefine any operand!
- How does information propagate? from top to bottom.
- How do we join info after tests?

Some defs

Definition

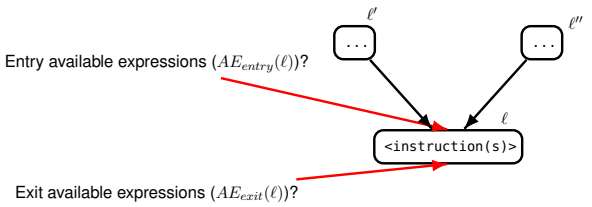
An expression is **killed** in a block if any of its variables is defined in the block.

Definition

A **generated** expression is an expression evaluated in the block and none of its variables is killed in the block.

- Sets : $kill_{AE}(block)$ and $gen_{AE}(block)$

Dataflow equations for AE - available expressions

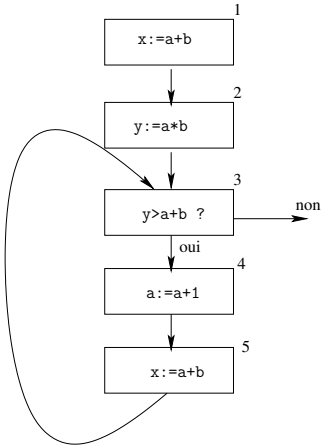


$$AE_{entry}(\ell) = \begin{cases} \emptyset & \text{if } \ell = init \\ \bigcap\{AE_{exit}(\ell') | (\ell', \ell) \in flow(G)\} \end{cases}$$

$$AE_{exit}(\ell) = \left(AE_{entry}(\ell) \backslash kill_{AE}(\ell)\right) \cup gen_{AE}(\ell)$$

On the example - equations

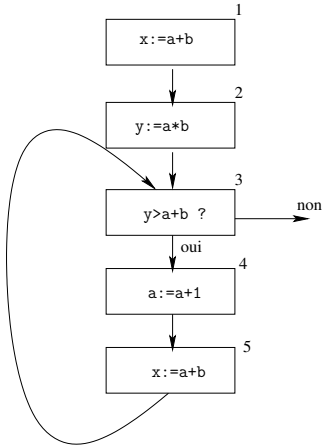
ℓ	$kill_{AE}(\ell)$	$gen_{AE}(\ell)$
1	$\emptyset$	{a+b}
2	$\emptyset$	{a∗b}
3	$\emptyset$	{a+b}
4	{a+b, a∗b, a+1}	$\emptyset$
5	$\emptyset$	{a+b}



On the example - final solution

ℓ	$AE_{entry}(\ell)$	$AE_{exit}(\ell)$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b, a∗b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}

- a+b is available on entry to the loop, not a∗b
- Improvement of code generation



Very Busy Expressions

```
var x, a, b

x = input

a = x - 1

b = x - 2

while (x > 0) {

    output a * b - x

    x = x - 1

}

output a * b
```

Consider the program on the left. How could we optimize it?

Which information does this optimization demand?

Very Busy Expressions

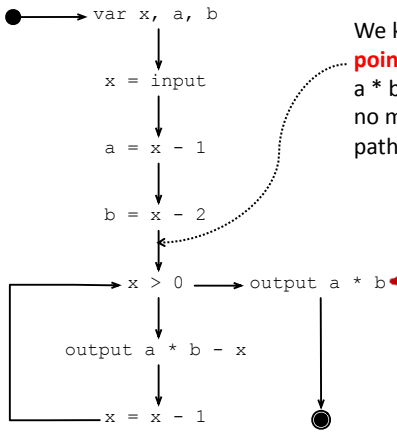
```
var x, a, b

x = input

a = x - 1

b = x - 2

while (x > 0) {

    output a * b - x

    x = x - 1

}

output a * b
```

- Consider the expression a * b
  - Does it change inside the loop?
- So, again, how could we optimize this program?

Which information does this optimization demands?

- Generally it is easier to see opportunities for optimizations in the CFG.
  - How is the CFG of this program?

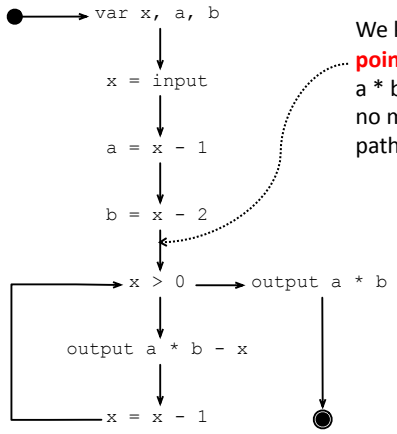
Very Busy Expressions



We know that at this **point**, the expression a * b will be computed no matter which program path is taken.

What if we did not have this command here?

Very Busy Expressions

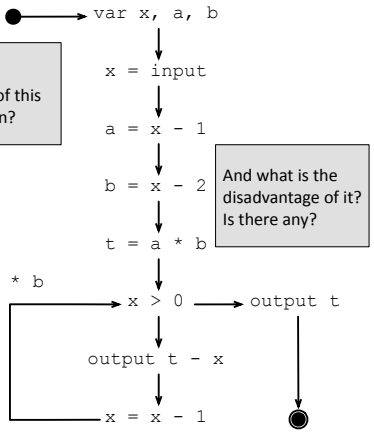


We know that at this **point**, the expression a * b will be computed no matter which program path is taken.

So, how can we improve this code?

Very Busy Expressions

What is the advantage of this optimization?

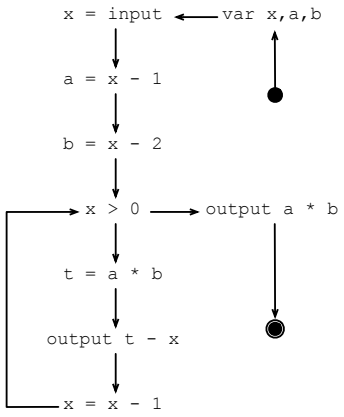


And what is the disadvantage of it? Is there any?

Very Busy Expressions

Which expressions are very busy in our example?

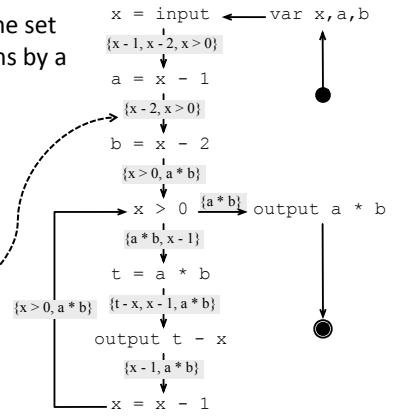
- In order to apply the previous optimization, we had to know that a * b was a *very busy expression* before the loop.
- An expression is very busy at a program point if it will be computed before the program terminates along any path that goes from that point to the end of the program.



Very Busy Expressions

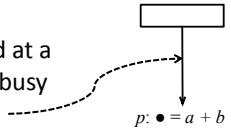
- We can approximate the set of very busy expressions by a dataflow analysis.
- How does information originate?
- How does information propagate?

Why is the expression a * b not very busy **here**?



The Origin of Information

If an expression is used at a point *p*, then it is very busy immediately before *p*.



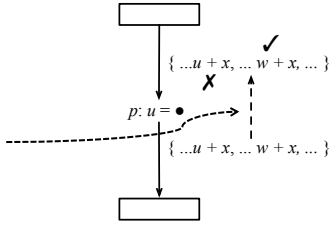
The Propagation of Information

- An expression *E* is very busy immediately before a program point *p* if, and only if:
  1. It is very busy immediately after *p*.
  2. No variable of E is redefined at *p*.

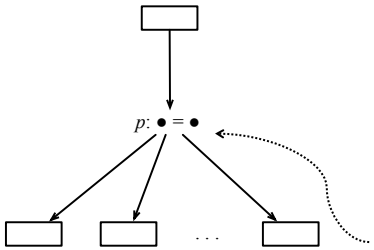
1. or
   1. It is used at *p*.

But, what if a program point has multiple successors?

This is the direction through which information propagates.



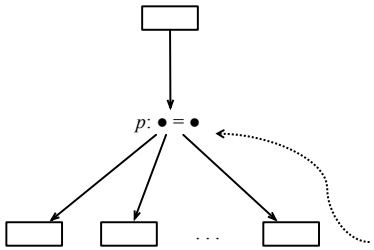
Joining Information



But, what if a program point has multiple successors?

How do we find the expressions that are very busy at this program point?

Joining Information

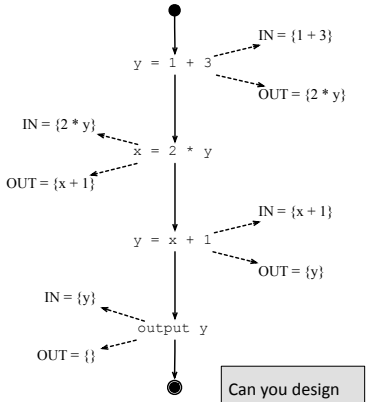


How do we find the expressions that are very busy at this program point?

If an expression E is very busy immediately before every successor of *p*, then it must be very busy immediately after *p*.

IN and OUT Sets for Very Busy Expressions

- To solve the very busy expression analysis, we associate with each program point *p* two sets, IN and OUT.
  - IN is the set of very busy expressions immediately before *p*.
  - OUT is the set of very busy expressions immediately after *p*.



Can you design equations to find these sets?

Dataflow Equations for Very Busy Expressions

$$p : v = E$$

$$IN(p) = (OUT(p) \setminus \{Expr(v)\}) \cup \{E\}$$

$$OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$$

- *IN(p)* = the set of very busy expressions immediately before *p*
- *OUT(p)* = the set of very busy expressions immediately after *p*
- *succ(p)* = the set of control flow nodes that are successors of p

How do these equations differ from those used in liveness analysis?

And what do they have in common?

Constant propagation

What kind of optimisation for:

```
x:=23;
y:=12+x;
while(y > x) do
   a:=a+x;
   y:=y-1;
done
```

- Propagate “i’m a constant” for each variable.
- Equations? Solving ?

The Dataflow Framework

- A may analysis keeps tracks of facts that *may* happen during the execution of the program.
  - A definition *may* reach a certain point.
- A must analysis tracks facts that *will* – for sure – happen during the execution of the program.
  - This expression *will* be used after certain program point.
- A backward analysis propagates information in the opposite direction in which the program flows.
- A forward analysis propagates information in the same direction in which the program flows.

The Dataflow Framework

	Backward	Forward
May	$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$ $OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$ Liveness	$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$ Reaching Defs
Must	$IN(p) = (OUT(p) \setminus \{v\}) \cup \{E\}$ $OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$ Very Busy Expressions	$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$ Available Expressions

Transfer Functions

	Backward	Forward
May	$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$ $OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$ Liveness	$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$ Reaching Defs
Must	$IN(p) = (OUT(p) \setminus \{v\}) \cup \{E\}$ $OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$ Very Busy Expressions	$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$ $OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$ Available Expressions

The transfer functions provides us with a new "interpretation" of the program. We can implement a machine that traverses the program, always fetching a given instruction, and applying the transfer function onto that instruction. This process goes on until the results produced by these transfer functions stop changing. This is abstract interpretation!

Transfer Functions

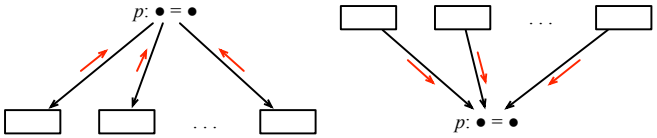
- The transfer functions do not have to be always the same, for every statement.
- In the concrete semantics of an assembly language, each statement does something different.
  - The same can be true for the abstract semantics of the programming language.

Statement	Transfer Function
exit	IN[p] = OUT[p]
output v	IN[p] = OUT[p] ∪ {v}
bgz v L	IN[p] = OUT[p] ∪ {v}
v = x + y	IN[p] = (OUT[p] \ {v}) ∪ {x, y}
v = u	IN[p] = (OUT[p] \ {v}) ∪ {u}

Which analysis is this one on the right?

The Merging Function

- The merging function specifies what happens once information collides♤.



How were the merging functions that we have seen so far?

♤: As we will see in the next classes, these functions have special names, e.g., meet and join.

Merging Functions

	Backward	Forward
May	$IN(p) = (OUT(p) \setminus \{v\}) \cup vars(E)$	$IN(p) = \bigcup OUT(p_s), p_s \in pred(p)$
	$OUT(p) = \bigcup IN(p_s), p_s \in succ(p)$	$OUT(p) = (IN(p) \setminus \{defs(v)\}) \cup \{p\}$
	Liveness	Reaching Defs
Must	$IN(p) = (OUT(p) \setminus \{v\}) \cup \{E\}$	$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$
	$OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$	$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$
	Very Busy Expressions	Available Expressions

The combination of transfer functions, merging functions and – to a certain extent – the way that we initialize the IN and OUT sets gives us guarantees that the abstract interpretation terminates.

The Dataflow Framework

- Many other program analyses fit into the dataflow framework.
  - Can you think of a few others?
- It helps a lot if we can phrase our analysis into this framework:
  - We gain efficient resolution algorithms (implementation).
  - We can prove termination (theory).
- Compiler writers have been doing this since the late 60's.

A Bit of History

- Dataflow analyses are some of the oldest allies of compiler writers.
- Frances Allen got the Turing Award of 2007. Some of her contributions touch dataflow analyses.

- Allen, F. E., "Program Optimizations", Annual Review in Automatic Programming 5 (1969), pp. 239-307
- Allen, F. E., "Control Flow Analysis", ACM Sigplan Notices 5:7 (1970), pp. 1-19
- Kam, J. B. and J. D. Ullman, "Monotone Data Flow Analysis Frameworks", Actal Informatica 7:3 (1977), pp. 305-318
- Kildall, G. "A Unified Approach to Global Program Optimizations", ACM Symposium on Principles of Programming Languages (1973), pp. 194-206