EISE 4 – Compilation

Quentin L. Meunier

Sorbonne Université, LIP6

2020

quentin.meunier@lip6.fr

Plan

1. **Compilation**
   - Introduction
   - Langage et grammaire
2. **Spécifications du projet**
3. **Ressources du projet**
4. **Déroulement du module**

Plan

1 Compilation
- Introduction
- Langage et grammaire

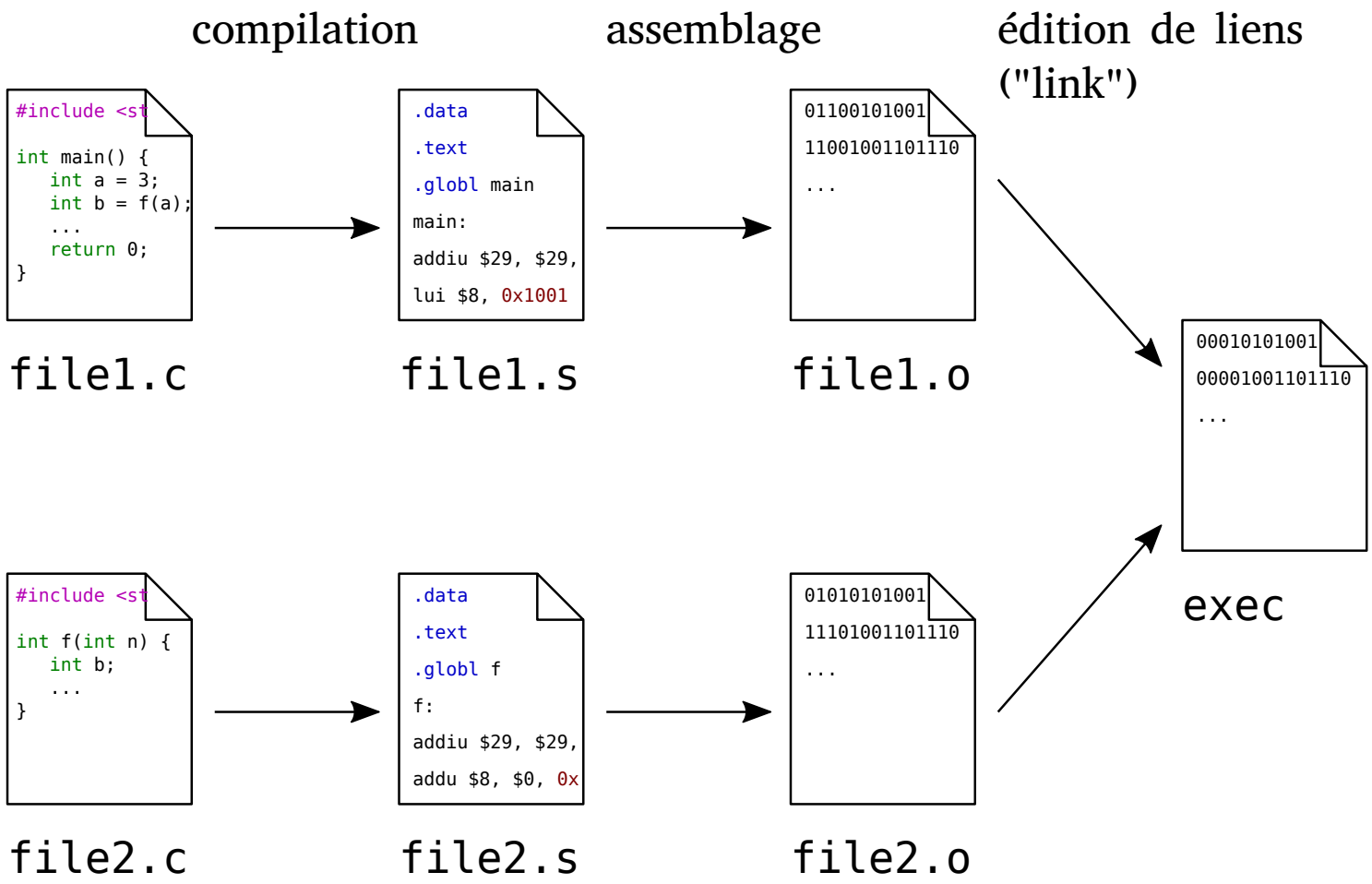
2 Spécifications du projet

3 Ressources du projet

4 Déroulement du module

La compilation

- **Objectif** : transformer un code source en binaire



La compilation

- Dans ce cours, focalisation sur la compilation à proprement parler, c'est-à-dire la transformation du code source en code assembleur
- $\Rightarrow$ **But du module** : écrire un compilateur
  - Projet sur 5 ou 6 séances, en binôme
- **Langage source** : sous-ensemble de C, appelé MiniC (avec quelques différences par rapport au C)
- **Langage cible** : assembleur Mips

Présentation informelle de MiniC

- Syntaxe du C
- 2 types de variables : `bool` et `int`
- Typage fort des expressions (pas de conversions implicite `int` $\rightarrow$ `bool`)
- Evaluation non-paresseuse des expressions
- Pas de :
  - Fonctions (sauf le `main`)
  - Pointeurs, tableaux
  - `switch`, `case`, `break`, `continue`, `goto`, labels
  - `typedef`, `struct`, `union`
  - `volatile`, `register`, `packed`, `inline`, `static`, `extern`
  - `unsigned`, `signed`, `long`, `long long`, `short`, `char`, `size_t`
  - `float`, `double`
  - Opérateurs `++`, `--`, `-=`, `+=`, `*=`, `/=`, `<<=`, `>>=`, `&=`, `|=`, ...
  - Cast
  - ...

Plan

1 Compilation
- Introduction
- Langage et grammaire

2 Spécifications du projet

3 Ressources du projet

4 Déroulement du module

Définition informelle d'un langage

- Un **langage** est un ensemble de mots sur un alphabet
- Exemple, sur l'alphabet { 'a', 'b' }, l'ensemble des mots { a, abb, baa, bbaa, aaaba } constitue un langage
- Un langage peut contenir un nombre infini de mots
- $\Rightarrow$ On ne peut pas décrire l'ensemble des mots de manière explicite, il faut un moyen inductif, comme une grammaire

Grammaire

- Une **grammaire** définit un langage
- Une grammaire contient les éléments suivants :
  - Un ensemble de **terminaux** V_T (aussi appelés *tokens*) : ce sont les éléments atomiques des mots du langage ; par exemple : { 'a', 'b', 'c' }
  - Un ensemble de **non-terminaux** V_N, par exemple { 'I', 'A', 'B' }
  - Un **axiome** (élément initial), qui est un non-terminal, par exemple `I`
  - Un ensemble de **règles de dérivation** qui permettent de "transformer" ce qu'il y a en partie gauche de la règle en ce qu'il y a en partie droite
- **Remarques** :
  - V_T et V_N sont appelés des **vocabulaires** et sont disjoints
  - Le vocabulaire de la grammaire est $V = V_T \cup V_N$
  - L'alphabet du langage induit par une grammaire est le vocabulaire terminal de la grammaire

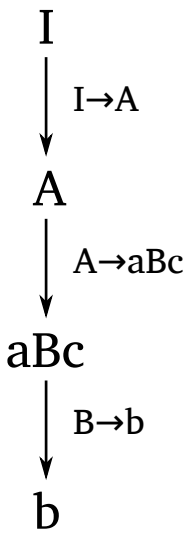
Grammaire : Exemple

- Soit une grammaire $G =< \{a, b, c\}, \{I, A, B\}, I, R >$ avec l'ensemble R de règles suivantes :
  - `I` $\rightarrow$ `A`
  - `I` $\rightarrow$ `BA`
  - `A` $\rightarrow$ `aBc`
  - `B` $\rightarrow$ `bB`
  - `B` $\rightarrow$ `b`
- Les mots `abc`, `abbc`, `bbabbc` appartiennent au langage engendré
- Les mots `bb`, `bac`, `abca`, `aabc` n'y appartiennent pas
- **Remarque** : le terminal ε désigne un élément vide ; par exemple, si on ajoute la règle `A` $\rightarrow \varepsilon$, le mot `bb` appartient au langage

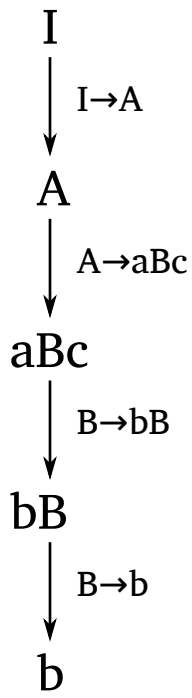
Arbre de dérivation

- Un **arbre de dérivation** représente les règles de dérivation qui sont prises à partir de l'axiome pour construire un mot du langage (une branche représente une règle prise)
- Arbres pour les exemples précédents :

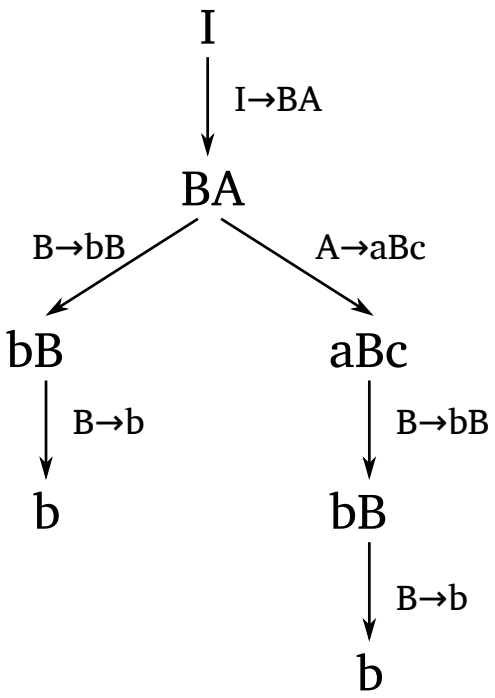
abc



abbc



bbabbc



Ambiguïté

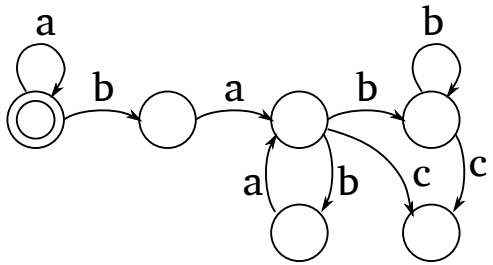
- Une grammaire est dite **ambigüe** quand un mot du langage peut être obtenu par au moins deux arbres de dérivations différents
- La grammaire précédente n'était pas ambigüe, tandis que la grammaire suivante l'est :
  - `I` $\rightarrow$ `AC`
  - `A` $\rightarrow$ `abA`
  - `A` $\rightarrow$ `a`
  - `C` $\rightarrow$ `baC`
  - `C` $\rightarrow$ `b`
- Le mot `abab` peut être obtenu de deux manière différentes
- On cherche en général à éviter les grammaires ambigües (la plupart des langages peuvent être décrits par une grammaire non-ambigüe)

Types de grammaires et de langages

- Les grammaires sont catégorisées selon la forme de leur règles
- $\Rightarrow$ Plus les règles sont contraintes, plus les analyses automatiques sont faciles, mais moins le langage est expressif
- Soit V_N l'ensemble des non-terminaux d'une grammaire, V_T l'ensemble des terminaux, et $V = V_T \cup V_N$
- Dans les définitions suivantes, $A, B \in V_N$, $\omega \in V_T^*$, $\psi \in V^+$, $\alpha, \beta \in V^*$
- Une grammaire est dite **régulière** si toutes ses règles sont de l'une des formes suivantes :
  - $A \rightarrow \omega B$
  - $A \rightarrow B\omega$
  - $A \rightarrow \omega$
- Une grammaire est dite **hors-contexte** si toutes ses règles sont de la forme :
  - $A \rightarrow \alpha$
- Une grammaire est dite **sous-contexte** si toutes ses règles sont de la forme :
  - $\alpha A \beta \rightarrow \alpha \psi \beta$
- Une grammaire est dite **générale** si ses règles sont de la forme :
  - $\alpha \rightarrow \beta$

Types de grammaires et de langages

- Si l'on ne considère pas les règles de la forme $A \rightarrow \varepsilon$, on a :
  grammaires régulières $\subset$ grammaires hors-contexte $\subset$ grammaires sous-contexte $\subset$ grammaires générales
- Les langages résultant des **grammaires régulières** sont équivalents aux langages décrits par des **expressions régulières**, et aux langages décrits par les **automates**
- **Exemple** :

Grammaire	Expression Régulière	Automate
A → aA | B B → baC C → baC | D D → bD | c	a*(ba)⁺b*c	

Types de grammaires et de langages

- Les langages réguliers sont les plus faciles à analyser, mais on ne peut pas tout exprimer ; par exemple, le langage $a^n b^n$ ne peut pas être décrit par un langage régulier, mais s'écrit trivialement avec une grammaire hors-contexte : $A \rightarrow aAb|\varepsilon$
- De même, le langage des parenthèses s'écrit facilement avec une grammaire hors-contexte mais ne peut pas s'écrire avec un langage régulier
- **Remarque** : la syntaxe des langages de programmation est souvent décrite à l'aide d'une grammaire hors-contexte
- **Pour aller plus loin** : polycopié de référence en théorie des langages : `http://lig-membres.imag.fr/mechenim/wp-content/uploads/sites/219/2016/05/PolycopieTL1.pdf`
(disponible aussi sur mon compte)

Plan

1. Compilation

2. Spécifications du projet
   - Introduction
   - Analyse lexicographique
   - Analyse syntaxique
   - Vérifications contextuelles
   - Génération de code
   - Autres

3. Ressources du projet

4. Déroulement du module

Plan

1 Compilation

2 Spécifications du projet
- Introduction
- Analyse lexicographique
- Analyse syntaxique
- Vérifications contextuelles
- Génération de code
- Autres

3 Ressources du projet

4 Déroulement du module

Introduction

- 4 étapes lors de la compilation :
  - Analyse lexicographique
  - Analyse syntaxique
  - Vérifications contextuelles
  - Génération de code

Exemple introductif : programme

- Soit le programme MiniC suivant :

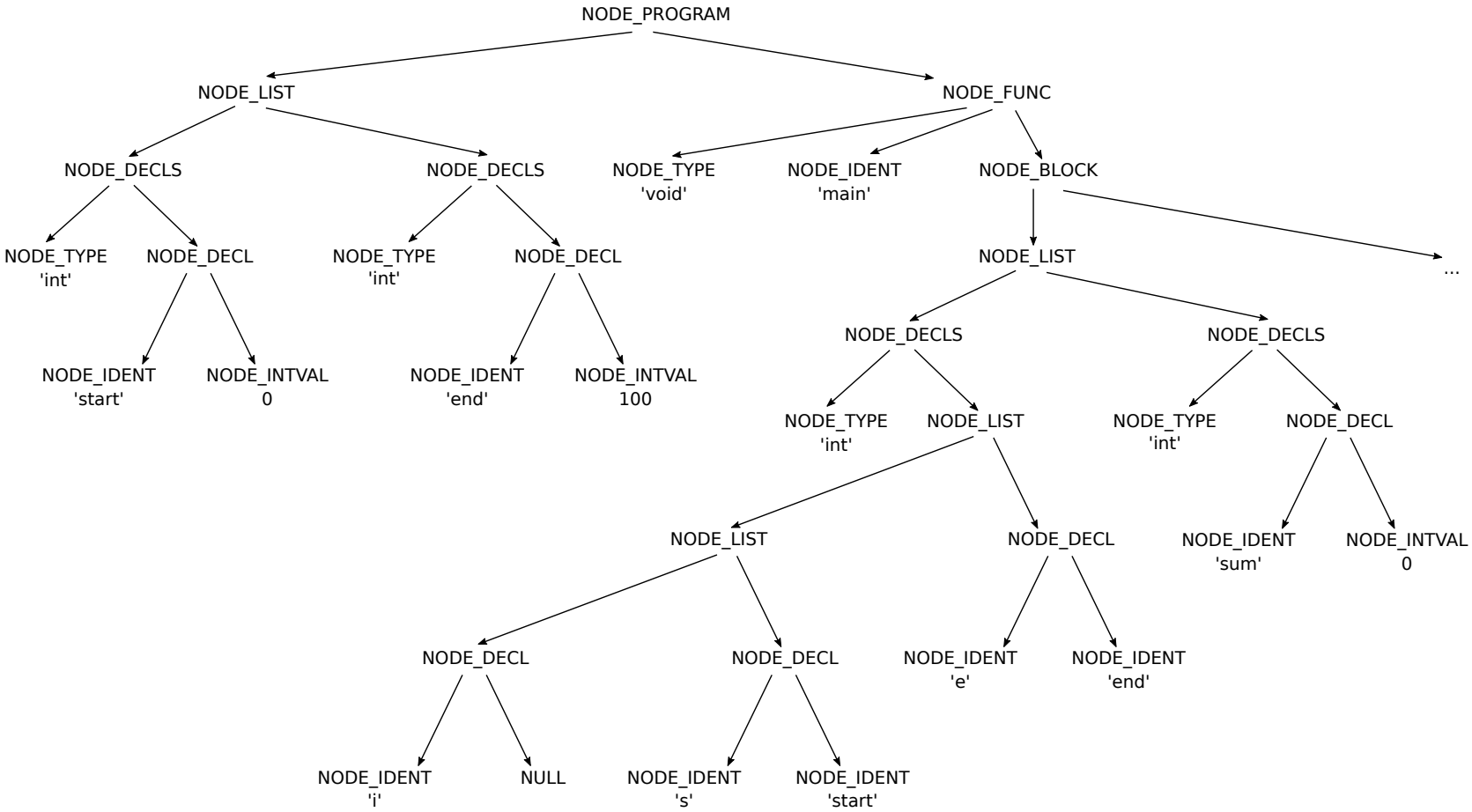
```
1   // Un exemple de programme MiniC
2   int start = 0;
3   int end = 100;
4
5   void main() {
6       int i, s = start, e = end;
7       int sum = 0;
8       for (i = s; i < e; i = i + 1) {
9           sum = sum + i;
10      }
11      print("sum: ", sum, "\n");
12  }
```

Exemple introductif : après analyse lexicographique

TOK_INT 2	TOK_IDENT 2 'start'	TOK_AFFECT 2	TOK_INTVAL 2 '0'	TOK_SEMICOL 2	TOK_INT 3
TOK_IDENT 3 'end'	TOK_AFFECT 3	TOK_INTVAL 3 '100'	TOK_SEMICOL 3	TOK_VOID 5	TOK_IDENT 5 'main'
TOK_LPAR 5	TOK_RPAR 5	TOK_LACC 5	TOK_INT 6	TOK_IDENT 6 'i'	TOK_COMMA 6	TOK_IDENT 6 's'
TOK_AFFECT 6	TOK_IDENT 6 'start'	TOK_COMMA 6	TOK_IDENT 6 'e'	TOK_AFFECT 6	TOK_IDENT 6 'end'
TOK_SEMICOL 6	TOK_INT 7	TOK_IDENT 7 'sum'	TOK_AFFECT 7	TOK_INTVAL 7 '0'	TOK_SEMICOL 7
TOK_FOR 8	TOK_LPAR 8	TOK_IDENT 8 'i'	TOK_AFFECT 8	TOK_IDENT 8 's'	TOK_SEMICOL 8
TOK_IDENT 8 'i'	TOK_LT 8	TOK_IDENT 8 'e'	TOK_SEMICOL 8	TOK_IDENT 8 'i'	TOK_AFFECT 8
TOK_IDENT 8 'i'	TOK_PLUS 8	TOK_INTVAL 8 '1'	TOK_SEMICOL 8	TOK_RPAR 8	TOK_LACC 8

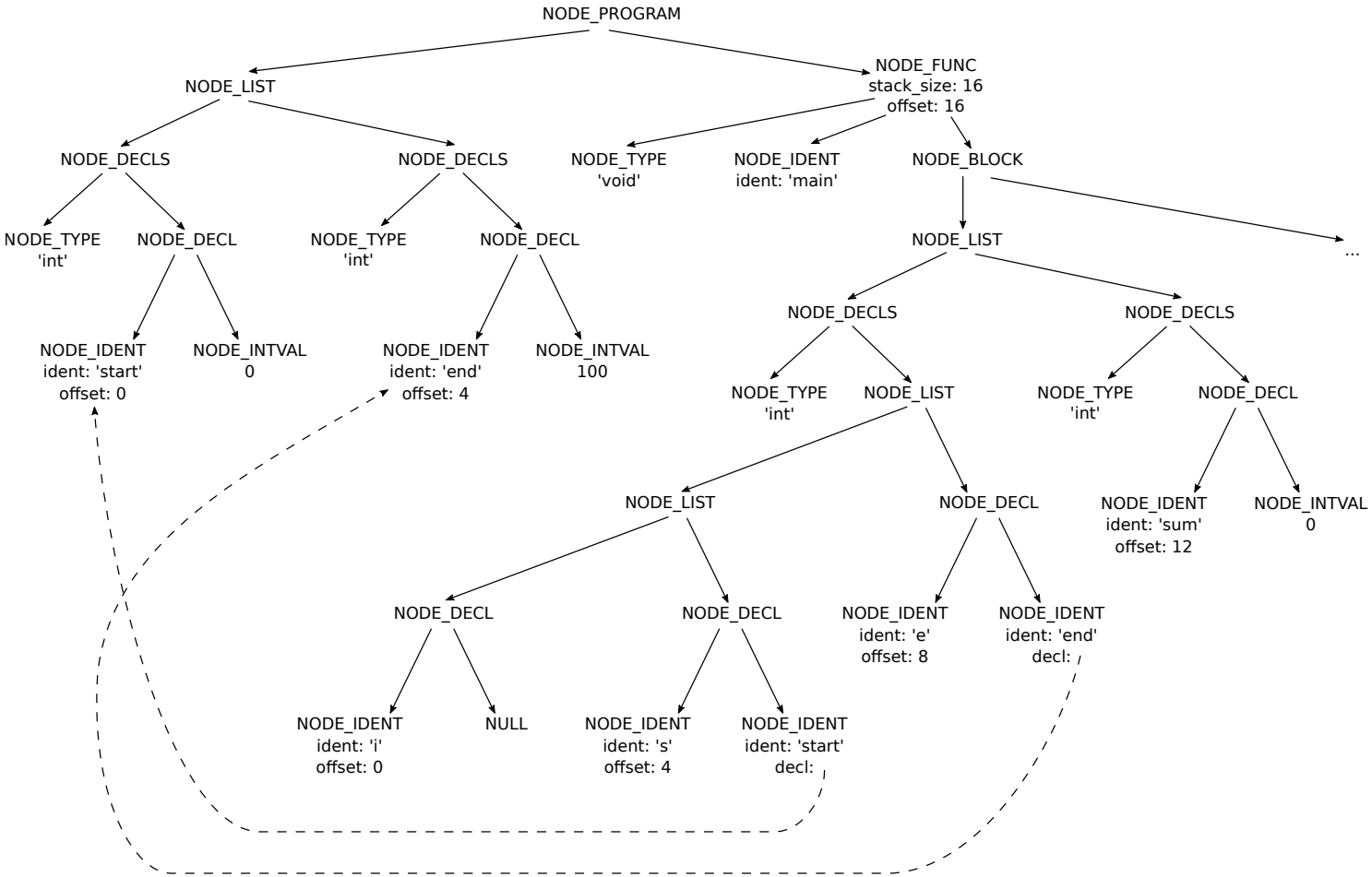
Exemple introductif : après analyse syntaxique

- Arbre (partie) du programme obtenu après analyse syntaxique



Exemple introductif : après vérifications contextuelles

- Arbre (partie) du programme obtenu après vérifications



Exemple introductif : après génération de code

- Code assembleur mips obtenu (1/2)

```
# Declaration des
# variables globales
.data

start: .word 0
end:   .word 100
.asciiz "sum: "
.asciiz "\n"

# Programme
.text

main:
    # Prologue : allocation en pile
    # pour les variables locales
    # i se trouve a l'adresse 0($29)
    # s se trouve a l'adresse 4($29)
    # e se trouve a l'adresse 8($29)
    # sum se trouve a l'adresse 12($29)
    addiu $29, $29, -16
```

```
# s = start
lui   $8, 0x1001
lw    $8, 0($8)
sw    $8, 4($29)
# e = end
lui   $8, 0x1001
lw    $8, 4($8)
sw    $8, 8($29)
# sum = 0
ori   $8, $0, 0
sw    $8, 12($29)
# for (i = s; i < e; i = i + 1)
# i = s
lw    $8, 4($29)
sw    $8, 0($29)
```

Exemple introductif : après génération de code

- Code assembleur mips obtenu (2/2)

```
    # i < e ?
_L1:
    lw      $8, 0($29)
    lw      $9, 8($29)
    slt     $8, $8, $9
    beq     $8, $0, _L2
    # sum = sum + i
    lw      $8, 12($29)
    lw      $9, 0($29)
    addu    $8, $8, $9
    sw      $8, 12($29)
    # i = i + 1
    lw      $8, 0($29)
    ori     $9, $0, 1
    addu    $8, $8, $9
    sw      $8, 0($29)
    # Retour au test de boucle
    j _L1
```

```
_L2:
    # print("sum :")
    lui     $4, 0x1001
    ori     $4, $4, 8
    ori     $2, $0, 4
    syscall
    # print(sum)
    lw      $4, 12($29)
    ori     $2, $0, 1
    syscall
    # print("\n");
    lui     $4, 0x1001
    ori     $4, $4, 14
    ori     $2, $0, 4
    syscall
    # Desallocation des variables
    # locales en pile
    addiu $29, $29, 16
    # exit
    ori     $2, $0, 10
    syscall
```

Plan

1 Compilation

2 Spécifications du projet
- Introduction
- Analyse lexicographique
- Analyse syntaxique
- Vérifications contextuelles
- Génération de code
- Autres

3 Ressources du projet

4 Déroulement du module

Première phase : analyse lexicographique

- Transformation du programme source en liste de tokens (terminaux du langage)
- Réalisé à partir de la lecture des caractères un par un par une machine d'état
- Pénible à faire $\Rightarrow$ outils pour générer cette machine à partir d'une description de plus haut niveau des tokens
- Outil communément utilisé : lex
- $\rightarrow$ C'est l'outil utilisé pour le projet

Lexicographie de MiniC

- Mots-clés du langage : void, int, bool, true, false, if, else, while, for, do, print
- Identificateurs (nom de variables) :
  - `LETTRE = {'a', ..., 'z', 'A', ..., 'Z'}`
  - `CHIFFRE = {'0', ..., '9'}`
  - `IDF = (LETTRE)(LETTRE | CHIFFRE | '_')*`
- Littéraux entiers
  - `CHIFFRE_NON_NUL = {1, ..., 9}`
  - `ENTIER_DEC = '0' | CHIFFRE_NON_NUL CHIFFRE*`
  - `LETTRE_HEXA = {'a', ..., 'f', 'A', ..., 'F'}`
  - `ENTIER_HEXA = '0x'(CHIFFRE | LETTRE_HEXA)+`
  - `SIGNE = '-' | ''`
  - `ENTIER = SIGNE(ENTIER_DEC | ENTIER_HEXA)`
- Chaines de caractère littérales
  - `CHAINE = '"'(CHAINE_CAR | '\"' | '\n')'"'`
  - Dans laquelle `CHAINE_CAR` est l'ensemble de tous les caractères imprimables, à l'exception de `'"'` et `'\'`

Lexicographie de MiniC

- Symboles spéciaux : il y a un certain nombre de symboles spéciaux qui ont chacun leur propre token associé : +, −, {, ...
  - → Voir la spécification complète pour l'exhaustivité de ces symboles
- Commentaires : suite de caractères imprimables et de tabulations qui commence par '//' et s'étend jusqu'à la fin de la ligne
  - Pas de token associé : une fois que l'on a détecté cette séquence, rien à renvoyer
- Les séparateurs de MiniC sont ' ' (caractère d'espace), tabulation horizontale et fin de ligne
  - Ce ne sont pas des tokens en eux-mêmes : ils servent à séparer les tokens

Lexicographie de MiniC : utilisation de lex

- Un fichier lex a le format suivant :

%{
Includes C et déclarations de fonctions
Copié tel quel dans le fichier produit par lex
%}
Définitions
%%
Règles
%%
Fonctions C (par exemple, main)
Copié tel quel dans le fichier produit par lex

Lexicographie de MiniC : utilisation de lex

- Définitions : de la forme `NOM expression`
- Exemples :
  - `LETTRE [A-Za-z]`
  - `IDF {LETTRE}({LETTRE}|{CHIFFRE}|_)*`
- Règles : de la forme `Caractères action`
- Exemples :
  - `"void" return TOK_VOID;`
  -
```
{IDF} {
        yylval.strval = strdup(yytext);
        return TOK_IDENT;
}
```
- Squelette du fichier fourni, à compléter

Plan

1 Compilation

2 Spécifications du projet
- Introduction
- Analyse lexicographique
- Analyse syntaxique
- Vérifications contextuelles
- Génération de code
- Autres

3 Ressources du projet

4 Déroulement du module

Deuxième phase : analyse syntaxique

- Transformation de la suite de tokens en arbre du programme
- Principe :
  - Retrouver les règles prises dans la grammaire (hors-contexte) du langage à partir de la suite de tokens
  - Les réduire dès que possible, i.e. “remonter” le non-terminal de la règle
  - Créer la ou les branches correspondantes dans l'arbre du programme
- **Remarque** : l'arbre de dérivation correspondant au programme et l'arbre du programme sont différents, i.e. toutes les règles de la grammaire ne se traduisent pas par une branche dans l'arbre du programme

Deuxième phase : analyse syntaxique

- Comme pour l'analyse lexicographique, le programme qui retrouve la règle à partir des tokens est généré à partir d'une description de la grammaire
- Utilise une **pile** de tokens :
  - Le token lu est mis au sommet de la pile
  - Si les n premiers tokens au sommet de la pile se réduisent en une règle, remplacement de tous ces tokens avec le non-terminal correspondant (au sommet de la pile) : **reduce**
  - Sinon, lecture du token suivant : **shift**
    - En réalité un petit peu plus compliqué car il faut considérer la priorité des opérateurs : il faut lire un token de plus avant de décider
  - On continue jusqu'à une réduction à l'axiome de la grammaire ; si on n'y arrive pas, le programme est syntaxiquement incorrect
- Outil communément utilisé : yacc (interface prévue avec lex)
- $\rightarrow$ C'est l'outil utilisé pour le projet

Syntaxe de MiniC

- Définie par une grammaire hors-contexte
- Priorité et associativité des opérateurs (exemples)
- La grammaire du langage est entièrement donnée dans le document de ressources
- $\Rightarrow$ Il faut faire la construction de l'arbre du programme

Syntaxe de MiniC : utilisation de yacc

- Un fichier yacc a le format suivant :

%{
Includes C et déclarations de fonctions
Copié tel quel dans le fichier produit par lex
%}
Définitions
%%
Règles
%%
Fonctions C
Copié tel quel dans le fichier produit par yacc

Syntaxe de MiniC : utilisation de yacc

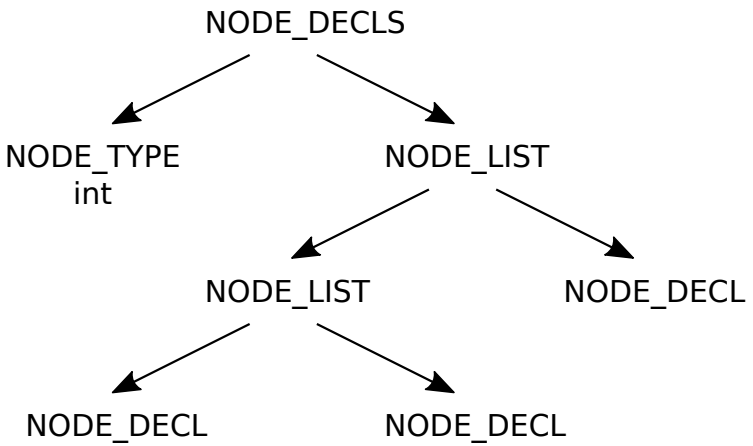
- La partie **définitions** contient principalement les déclarations des tokens, leur priorité et leur associativité
  - `%left`, `%right` ou `%nonassoc`
  - Du moins prioritaire vers le plus prioritaire
  - **Exemple** : `%left TOK_OR`
- Définit aussi le type retourné par les tokens ayant des informations supplémentaires (ex : littéral) et par les non-terminaux (noeud de l'arbre)
- **Exemples** :
  - `%type <intval> TOK_INTVAL`
  - `%type <ptr> program`

Syntaxe de MiniC : utilisation de yacc

- La partie **règles** contient les règles de la grammaire du langage et les actions associées
  - Les actions sont entre accolades
  - `$$` représente ce qui est retourné (un noeud de l'arbre)
  - `$i` représente ce qui est retourné par le i-ème élement (terminal ou non) en partie droite de la règle
- **Exemple** :
  - `expr : expr TOK_MUL expr { $$ = make_node(NODE_MUL, 2, $1, $3); }`
  - `make_node` est une fonction écrite dans la dernière partie du fichier, et prenant un nombre variable de paramètres

Syntaxe de MiniC : cas des noeuds liste

- Dans certains cas, les règles sont récursives, pour traduire le fait que le programme contient une succession d'éléments
  - **Exemple** : instructions, déclaration des variables
- Au niveau de l'arbre, on implémente cela en utilisant des noeuds particuliers, appelés noeuds liste (`NODE_LIST`)
- **Exemple** : `int a, b, c` produit l'arbre :



- Règles correspondantes :

```
listtypedecl  : decl                          { $$ = $1; }
              | listtypedecl TOK_COMMA decl { $$ = make_node(NODE_LIST, 2, $1, $3); }
              ;
```

Syntaxe de MiniC : arbres corrects

- L'ensemble des arbres de programme corrects est défini par une grammaire, appelée **grammaire d'arbres**
- Cette grammaire définit le nombre et les types des noeuds enfants que peuvent avoir les noeuds d'un certain type
- Cette grammaire est entièrement spécifiée dans le document de spécification

Plan

1 Compilation

2 Spécifications du projet
- Introduction
- Analyse lexicographique
- Analyse syntaxique
- Vérifications contextuelles
- Génération de code
- Autres

3 Ressources du projet

4 Déroulement du module

Vérifications contextuelles

- Un programme syntaxiquement correct n'est pas forcément correct
- **Exemples** : référence à une variable non déclarée, types des opérandes d'un opérateur incompatibles (`bool` + `bool`)
- Vérifier que le programme est correct nécessite une passe spécifique : la passe de **vérification contextuelle**
- Toutes les vérifications sont spécifiées formellement par une grammaire attribuée : elle vous est donnée pour ce projet
- La passe doit implémenter ces vérifications
- La passe de vérifications permet aussi de rattacher les noeuds d'occurrence des variables à leur définition

Grammaire attribuée

- **Objectif** : Pouvoir lire la grammaire attribuée ; décodage des règles et vérifications à faire vous-mêmes
- Grammaire attribuée : grammaire concrète du langage, enrichie d'**attributs** (donnée ou structure de données : ensemble, type, etc.)
- Les attributs peuvent être soit **hérités** ($\downarrow$), soit **synthétisés** ($\uparrow$)
- Un attribut hérité "descend" l'arbre de dérivation :
  - Il peut être lu s'il apparait en partie gauche de la règle
  - Il doit être affecté s'il apparait dans la partie droite de la règle
- Un attribut synthétisé "remonte" l'arbre de dérivation :
  - Il peut être lu s'il apparait en partie droite de la règle
  - Il doit être affecté s'il apparait en partie gauche de la règle

Grammaire attribuée

- **Affectation** : explicite ou implicite
  - Explicite : clause *affectation*

$$\begin{array}{lll} \textbf{ident} \downarrow env \uparrow type & \rightarrow & \underline{\textbf{idf}} \uparrow nom \\ & \textbf{affectation} & type := env(nom) \rightarrow type \end{array}$$

  - Implicite : valeur d'affectation directement écrite à côté de l'attribut

$$\textbf{programme} \rightarrow \textbf{main\_declaration} \downarrow \{\}$$

Grammaire attribuée

- **Condition** : explicite ou implicite
  - Explicite : clause *condition*

$$\begin{array}{rl} \textbf{decl\_var} \downarrow env \downarrow ctx \downarrow type \downarrow global \uparrow ctx \cup \{nom \mapsto def(type, ...)\} & \\ \rightarrow & \underline{\textbf{idf}} \uparrow nom \\ \textbf{condition} & nom \notin dom(ctx) \end{array}$$

  - Implicite : par filtrage

$$\textbf{inst} \downarrow env \rightarrow \underline{\textbf{while}}\ \text{'('}\ \textbf{exp} \downarrow env \uparrow \underline{bool}\ \text{')'}\ \textbf{bloc} \downarrow env$$

Grammaire attribuée

- **Abbréviation** : éviter d'introduire des noms inutiles quand les attributs ne sont pas utilisés

  param_print $\downarrow env \quad \rightarrow \quad$ **ident** $\downarrow env \uparrow type$

  peut s'écrire :

  param_print $\downarrow env \quad \rightarrow \quad$ **ident** $\downarrow env \uparrow \_$

Contexte et environnement

- Un **contexte** contient un ensemble d'associations : nom (de variable) $\rightarrow$ définition
  - $\sim$ Structure de données de type `map` en programmation (clé, valeur)
- $\Rightarrow$ Un contexte ne peut contenir qu'une fois un nom donné
- Un **environnement** est un empilement (une pile) de contextes
- La définition associée à une variable est cherchée dans le contexte au sommet, puis (si définition absente), dans le contexte suivant (contexte dessous le sommet de pile), etc.
- Une définition récente d'une variable (contexte en haut de la pile) masque une définition plus ancienne (contexte en bas de la pile)

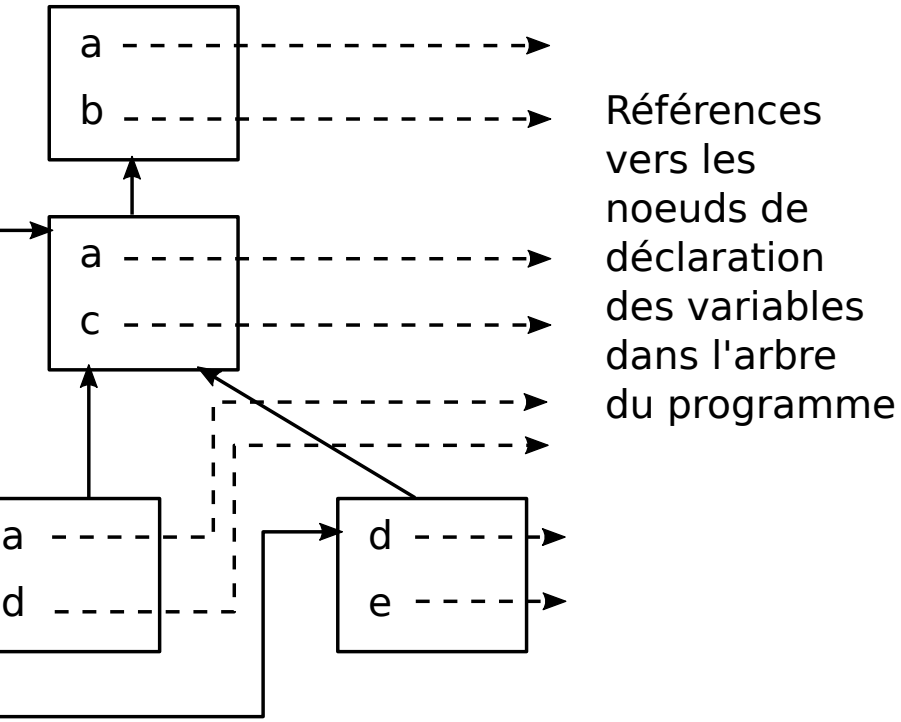
Contexte et environnement

- **Exemple** : environnement d'analyse des différents blocs d'un programme

```
int a = 0;
int b = 0;

void main() {
   int a = 1;
   int c = 2;

   if (true) {
      int a = 5;
      int d = 6;
      a = a + b + c + d;
   }
   else {
      int d;
      int e;
      e = d = 1;
   }
}
```



Grammaire attribuée de MiniC : exemples

- **Exemple** : déclaration de variables

$$\begin{aligned} \textbf{decl\_vars}\ {\downarrow}env\ {\downarrow}ctx_0\ {\downarrow}global\ {\uparrow}ctx_1 \\ \rightarrow\quad & \textbf{type}\ {\uparrow}type \\ & \textbf{liste\_declarations\_type}\ {\downarrow}env\ {\downarrow}ctx_0\ {\downarrow}type \\ & {\downarrow}global\ {\uparrow}ctx_1\ \text{';'} \\ \textbf{condition}\quad & type \neq \underline{void} \end{aligned}$$

- **Exemple** : boucle `for`

$$\begin{aligned} \textbf{inst}\ {\downarrow}env \rightarrow\quad & \underline{\textbf{for}}\ \text{'('}\ \textbf{exp}\ {\downarrow}env\ {\uparrow}\_\ \text{';'}\ \textbf{exp}\ {\downarrow}env\ {\uparrow}\underline{bool} \\ & \text{';'}\ \textbf{exp}\ {\downarrow}env\ {\uparrow}\_\ \text{')'}\ \textbf{inst}\ {\downarrow}env \end{aligned}$$

Grammaire attribuée : exemples

- **Exemple** : bloc

$$\textbf{bloc} \downarrow env \quad \rightarrow \quad \text{'\{'} \ \textbf{liste\_declarations} \downarrow env \downarrow \{\} \downarrow false \uparrow ctx \ \textbf{liste\_inst} \downarrow ctx/env \ \text{'\}'}$$

- a/b dénote l'environnement obtenu par l'empilement du contexte a sur l'environnement b

Plan

1 Compilation

2 Spécifications du projet
- Introduction
- Analyse lexicographique
- Analyse syntaxique
- Vérifications contextuelles
- Génération de code
- Autres

3 Ressources du projet

4 Déroulement du module

Génération de code

- **But de la passe** : produire le programme assembleur Mips correspondant à un arbre vérifié
- Parcours de l'arbre en profondeur, génération des instructions dans l'ordre du programme
  - La génération des instructions du noeud courant doit être faite après celles de ses fils
- $\rightarrow$ Exemples à la prochaine séance (TD)

Plan

1 Compilation

2 Spécifications du projet
- Introduction
- Analyse lexicographique
- Analyse syntaxique
- Vérifications contextuelles
- Génération de code
- Autres

3 Ressources du projet

4 Déroulement du module

Ligne de commande du compilateur

- Le compilateur doit gérer un certain nombre d'**options** sur la ligne de commande, en particulier :
  - Une option pour limiter le nombre de registres utilisés
  - Une option pour stopper la compilation après l'analyse syntaxique
  - Une option pour stopper la compilation après la passe de vérifications
  - Une option pour définir le **niveau de trace** ; par défaut (niveau 0), la compilation d'un code correct **ne doit rien afficher**
- Ces options seront utilisées par les scripts d'évaluation automatique $\Rightarrow$ Nécessité de les respecter
- Spécifications complètes et exemples dans le polycopié

Formattage des message d'erreur

- Les programmes source incorrects doivent lever une erreur lors de leur compilation : **erreur de compilation** (différent des erreurs internes au compilateur)
- Les différents types d'erreur de compilation doivent être **formatés** de la manière suivante :

  `Error line <num_ligne>: <decription du problème>`

  par exemple :

  `Error line 12: variable ‘‘foo’’ undeclared`
- Respecter ce format est très important : en particulier, la chaine `line <num>` sera recherchée automatiquement par les scripts d'évaluation, et le numéro de ligne vérifié
- S'arrêter à la première erreur rencontrée dans le programme source

Plan

1. Compilation
2. Spécifications du projet
3. Ressources du projet
   - Vue d'ensemble des modules à écrire
   - Modules fournis
   - Code fourni
   - Organisation du travail
4. Déroulement du module

Plan

1 Compilation

2 Spécifications du projet

3 Ressources du projet
- Vue d'ensemble des modules à écrire
- Modules fournis
- Code fourni
- Organisation du travail

4 Déroulement du module

Philosophie

- La quasi-totalité des modules sont à écrire
- **Principe** : voir la chaine de compilation de bout en bout
- Néanmoins, pour permettre à certains groupes d'avoir des résultats avant d'avoir tout écrit, **quelques implémentations fournies** : binaire + interface .h
- **Idée** : pouvoir avoir un compilateur qui marche même sans avoir tout écrit
  - $\rightarrow$ Bien sûr pris en compte dans la notation
  - Permet d'avoir plus facilement une approche incrémentale
- Libre de garder la même interface (fonctions, paramètres, etc.) ou de la changer dans vos propres implémentations
- **Attention** :
  - Utiliser les modules fournis peut influencer ou contraindre votre écriture du module par la suite
  - Binaires fournis pour linux (machines de l'école)
  - Expérimental...

Modules à écrire

- Analyse lexicographique : fichier lex à compléter
- Analyse syntaxique : fichier yacc à compléter
- Analyse des arguments de la ligne de commande et options
- Module de contexte
- Module d'environnement
- Allocateur de registres
- Première passe : vérifications contextuelles
- Deuxième passe : génération de code

Plan

1 Compilation

2 Spécifications du projet

3 Ressources du projet
- Vue d'ensemble des modules à écrire
- Modules fournis
- Code fourni
- Organisation du travail

4 Déroulement du module

Définitions générales au projet : fichier defs.h

- Définition du type `node_t` : noeud de l'arbre du programme
- Définition de l'enum `node_nature` : natures possibles pour un noeud
- Définition de l'enum `node_type` : type de l'expression associée au noeud
- Ce fichier ne devrait pas être modifié

Création des programmes mips

- Module **fourni** : pas à écrire
- Fournit la représentation d'un programme assembleur
- Fonctions pour créer les différents types d'instructions et directives mips
- Création du fichier final
- Module documenté dans le polycopié

Module de contexte

- Réalise l'association entre un nom de variable et sa définition (noeud associé à la déclaration dans l'arbre du programme)
- Plusieurs implémentations possibles, celle fournie utilise un arbre indexé par les caractère du nom (temps de recherche indépendant du nombre d'éléments)
- Type `context_t`

Module d'environnement

- Réalise la gestion de l'**empilement** et du **dépilement** des contextes
- Permet d'associer un nom de variable à sa définition dans le contexte le plus proche (interne)
- Chainage des contextes entre eux, type `env_t`
- **Difficulté** : calcul des offsets (en pile ou dans la section `.data`) des variables du programme

Module de l'allocateur de registres

- **Objectif** : déterminer le numéro des registres utilisés pour le calcul des expressions
- **Difficulté** : gérer le cas quand il n'y a plus de registres disponibles
- **Exemple** : `a = 1 + (2 + (3 + (4 + 5)));`
- Expressions évaluées **de gauche à droite**, mais priorité liée aux parenthèses
- Code assembleur possible (utilise 5 registres) :

```
addiu r8, r0, 1
addiu r9, r0, 2
addiu r10, r0, 3
addiu r11, r0, 4
addiu r12, r0, 5
addu  r11, r11, r12
addu  r10, r10, r11
addu  r9, r9, r10
addu  r8, r8, r9
sw    r8, 4(r29) # adresse de a
```

Module de l'allocateur de registres

- Si l'on ne dispose maintenant que de 4 registres ⇒ Nécessaire de stocker des valeurs intermédiaires en pile

```
addiu r8, r0, 1
addiu r9, r0, 2
addiu r10, r0, 3
sw    r10, 8(r29)
addiu r10, r0, 4
sw    r10, 12(r29)
addiu r10, r0, 5
```

```
lw    r11, 12(r29)
addu  r10, r11, r10
lw    r11, 8(r29)
addu  r10, r11, r10
addu  r9, r9, r10
addu  r8, r8, r9
sw    r8, 4(r29)
```

- ⇒ Il faut allouer deux mots de plus en pile au début de la fonction (en même temps que les variables locales)
- **Remarque** : Ici, une optimisation basée sur la propagation des constantes permettrait de charger directement la valeur 15 dans un registre, mais ce problème se pose plus sérieusement dès qu'il y a des expressions plus complexes contenant des effets de bord (exemple : appels de fonctions)

Module de l'allocateur de registres

- **Exemple** : si l'on enlève les parenthèses de l'expression précédente :
  `a = 1 + 2 + 3 + 4 + 5;`
- Besoin uniquement de deux registres :

```
addiu r8, r0, 1
addiu r9, r0, 2
addu r8, r8, r9
addiu r9, r0, 3
addu r8, r8, r9
```

```
addiu r9, r0, 4
addu r8, r8, r9
addiu r9, r0, 5
addu r8, r8, r9
sw r8, 4(r29)
```

- **Remarque** : il s'agit d'une implémentation naïve, qui peut s'optimiser en utilisant directement des instructions `addiu r8, r8, x` (pas demandé pour le projet)

Module de l'allocateur de registres : interface fournie

- L'interface fournie comporte beaucoup de fonctions
- $\Rightarrow$ À vous de voir si vous voulez investir du temps pour maitriser l'interface (peut aussi vous aider pour votre propre implémentation)
- Module complexe
- **Conseil** : dans un premier temps, faire un allocateur simple qui lève une erreur quand il n'y a plus de registre disponible

Affichage de l'arbre du programme

- Fonction `dump_tree()` fournie dans le fichier `common.c`
- Produit un graphe de l'arbre au format `graphviz`, visualisable avec `dot` (ou `xdot`)
- Utilisation libre, pratique pour le débug
- Customizable

Allocations et désallocations mémoire

- Votre compilateur devra désallouer toutes les structures allouées et ne contenir aucune fuite mémoire
- Vérifié lors de l'évaluation
- Pensez à utiliser `valgrind`
- De plus :
  - Il faut appeler la fonction `yylex_destroy()` à la fin de votre `main()`
  - Il faut compiler le fichier produit par yacc avec l'option `-DYY_NO_LEAKS`

Plan

1 Compilation

2 Spécifications du projet

3 Ressources du projet
- Vue d'ensemble des modules à écrire
- Modules fournis
- Code fourni
- Organisation du travail

4 Déroulement du module

Code fourni

- Fichiers : `lexico.l` (à compléter), `grammar.y` (à compléter), `common.c`, `common.h`, `libutils.a`
  - Sur les machines de l'école : à l'emplacement `~meunierq/projet_compilation_src.tar`
- Binaire `minicc` de référence, `xdot` et simulateur mips `Mars` (`Mars_4_2.jar`)
  - Sur les machines de l'école : à l'emplacement `~meunierq/bin`
- **Remarque** : un groupe qui trouve un bug dans le compilateur de référence (ce que je considère être un bug) gagne 1 point de bonus sur sa note finale

Plan

1 Compilation

2 Spécifications du projet

3 Ressources du projet
- Vue d'ensemble des modules à écrire
- Modules fournis
- Code fourni
- Organisation du travail

4 Déroulement du module

Gestion de projet

- Travail en binôme (pas de trinôme)
  - Répartition des tâches libre (entre les binômes et dans le temps), mais les deux binômes doivent avoir une bonne connaissance du code
- Commencer par la partie lex et yacc
- Affichage des chaines de caractère : devrait être fait assez tôt
  - Utile pour votre propre débug et pour l'évaluation
- Scripts de test, tests de non-régression
- Optimisations
- Modification des structures fournies

Rapport et livrables

- Rendre une archive au format .tar.gz contenant :
  - Le code (fichiers .c et .h)
  - Le ou les makefile(s)
  - Les scripts de tests
  - Les fichiers de test
  - Pas de binaire
- Écriture d'un rapport de 20 à 25 pages qui décrit la spécification des différents modules (packages) réalisés, ainsi que l'infrastructure de test
- Une soutenance aura éventuellement lieu lors de la dernière séance

Évaluation

- 40% : Passage automatisé de votre compilateur sur un ensemble de tests
- 20% : Qualité d'écriture de votre code (style, indentation, nommage des variables, découpage en fonctions pertinent)
- 20% : L'ensemble de vos tests et l'automatisation de ces test (évaluation automatique de vos tests)
- 10% : Rapport décrivant l'architecture logicielle
- 10% : Soutenance (si elle a lieu)

Fraude

- Expérience personnelle passée : nombreux cas et sanctions (plus d'une centaine en 12 ans)
- Extrait du règlement :
  En cas de fraude, l'élève est susceptible d'être déféré en section disciplinaire de l'établissement et s'expose aux sanctions suivantes :
  - *l'avertissement*
  - *le blâme*
  - *l'exclusion de l'établissement pour une durée maximum de 5 ans - cette sanction peut être prononcée avec sursis si l'exclusion n'excède pas 2 ans*
  - *l'exclusion définitive de l'établissement*
  - *l'exclusion de tout établissement public d'enseignement supérieur pour une durée maximum de 5 ans*
  - *l'exclusion définitive de tout établissement public d'enseignement supérieur.*
- Tout échange de code, y compris de fichiers de tests, entre deux binômes différents constitue une fraude et entrainera la note de 0 pour les deux membres des deux binômes et/ou la constitution d'un dossier auprès de l'instance compétente de l'université.
- Valable aussi pour les codes de l'année dernière : analyse automatique par des outils de recherche de plagiat
- Conséquence : protégez vos comptes et vos données

Plan

1. Compilation
2. Spécifications du projet
3. Ressources du projet
4. Déroulement du module

Déroulement du module

- **Conseil** : développement “transversal” aux modules : commencer par avoir la chaine complète (exception : lexicographie et syntaxe) pour des programmes simples, puis prendre en compte de plus en plus d'aspects du langages (expressions, chaines de caractère, variables globales, variables locales, structures de contrôle, etc.)
- $\Rightarrow$ **Spécification par le test**
- Avancement indicatif par semaine :
  - Lundi 23 Mars : TD
  - Lundi 30 Mars : Fichiers lex et yacc
  - Lundi 20 Avril : Ligne de commande, chaines de caractères
  - Lundi 27 Avril : Variables globales, expressions
  - Lundi 4 Mai : Variables locales et structures de contrôle
  - Lundi 11 Mai : Finalisation des modules
  - Lundi 18 Mai : Soutenance ?
- Tests en parallèle
- Travailler en dehors des séances de TP

Derniers conseils

- Lisez et re-lisez le polycopié
- Faites des tests
- Utilisez valgrind
- Rendre un code que vous n'avez pas écrit est rarement un bon pari