Hugues Cassé <casse@irit.fr>
M1 SIAME - FSI - Université de Toulouse

Cours 2 : Arbres de Syntaxe Abstraite

COMC

Introduction

Rôle du compilateur :

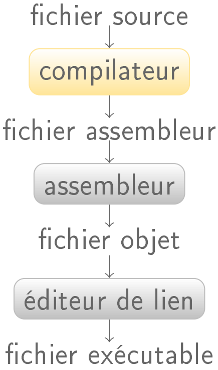
- générer le code (section `.text`)
- générer les données constantes (section `.rodata`)
- générer les variables (section `.data` et `.bss`)
- générer la table des symboles (section `.symtab`)

fichier source
↓
compilateur
↓
fichier assembleur
↓
assembleur
↓
fichier objet
↓
éditeur de lien
↓
fichier exécutable

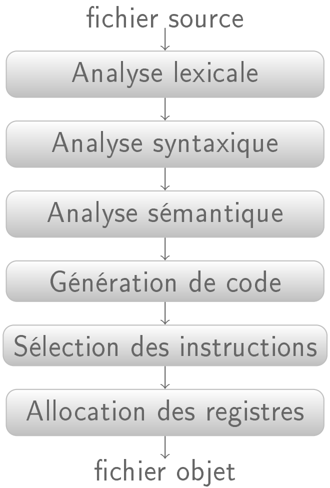
Introduction

Rôle du compilateur :

- générer le code (section `.text`)
- générer les données constantes (section `.rodata`)
- générer les variables (section `.data` et `.bss`)
- générer la table des symboles (section `.symtab`)



Organisation du compilateur




```c
int sum(int n) {
  int s = 0, i = 0;
  while(i <= n) {
    s += i;
    i++;
  }
  return s;
}
```

Organisation du compilateur

fichier source

↓

- **Analyse lexicale**
- Analyse syntaxique
- Analyse sémantique
- Génération de code
- Sélection des instructions
- Allocation des registres

↓

fichier objet

```
[
  TYPE(T_INT); ID("sum"); LPAREN;
  TYPE(T_INT); ID("n"); RPAR;
  LBRACE; TYPE(T_INT); ID("s");
  EQ, INT(0); COM; ID("i"); EQ;
  INT(0); SEMI;
  WHILE; LPAREN; ID("i"); LE;
  ID("n"); LBRACE; ID("s");
  PLUS_EQ; ID("i"); SEMI;
  ID("i"); PLUS_PLUS; RBRACE;
  RETURN; ID("s"); SEMI;
  RBRACE
]
```

Organisation du compilateur

fichier source

↓

Analyse lexicale

↓

Analyse syntaxique

↓

Analyse sémantique

↓

Génération de code

↓

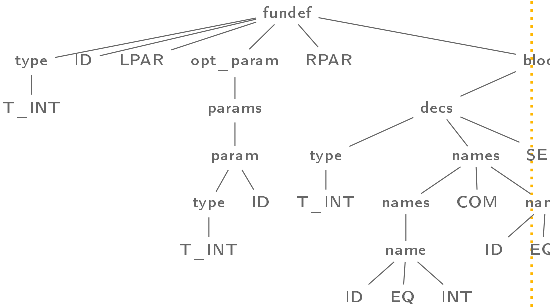
Sélection des instructions

↓

Allocation des registres

↓

fichier objet



Organisation du compilateur

fichier source

↓

Analyse lexicale

↓

Analyse syntaxique

↓

Analyse sémantique

↓

Génération de code

↓

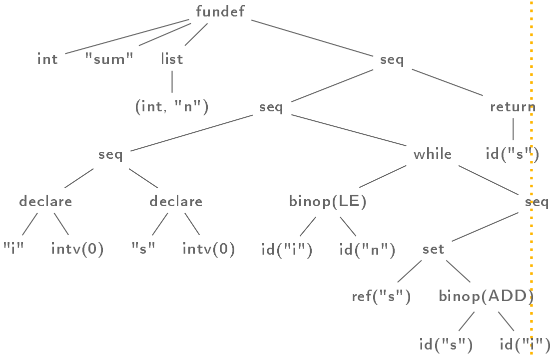
Sélection des instructions

↓

Allocation des registres

↓

fichier objet



Organisation du compilateur

fichier source

Analyse lexicale

Analyse syntaxique

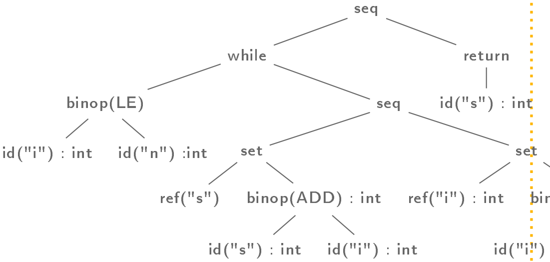
Analyse sémantique

Génération de code

Sélection des instructions

Allocation des registres

fichier objet



Plan

Objectifs

AST (*Abstract Syntactic Tree*) ou Arbre de Syntaxe Abstraite :

- représenter complètement le programme
- minimiser la représentation
- minimiser les constructeurs différentes
- facile à traiter

Situation :

- proche du code source (mais simplifiée)
- base pour la traduction en langage de bas niveau

Syntaxe abstraite

$$\begin{aligned} \text{entité} : \quad & \textit{constructeur}_1(\text{paramètre}^*) \\ & \mid \textit{constructuer}_2(\text{paramètre}^*) \\ & \mid \ldots \end{aligned}$$

$$\begin{aligned} \text{paramètre} : \quad & \text{entité} & \\ & \mid \textit{ensemble} & \mathbb{B}, \mathbb{N}, \mathbb{Z}, \mathbb{R} \\ & \mid \text{paramètre}^* & \textit{liste} \\ & \mid 2^{\text{paramètre}} & \textit{ensemble} \\ & \mid \text{paramètre} \times \text{paramètre} & \textit{paire} \end{aligned}$$

$\Longleftrightarrow$ description d'un arbre typé !

Instructions

$$
\begin{aligned}
S :\ & nop \\
\mid\ & set(R, E) \\
\mid\ & seq(S, S) \\
\mid\ & if(E, S, S) \\
\mid\ & while(E, S) \\
\mid\ & dowhile(S, E) \\
\mid\ & call(E, E^*) \\
\mid\ & return(E) \\
\mid\ & block(S) \\
\mid\ & declare(ID, T, S)
\end{aligned}
$$

$$
\begin{aligned}
R :\ & noref \\
\mid\ & id(ID)
\end{aligned}
$$

$$
\begin{aligned}
T :\ & void \\
\mid\ & int \\
\mid\ & char \\
\mid\ & float \\
\mid\ & fun(T, T^*)
\end{aligned}
$$

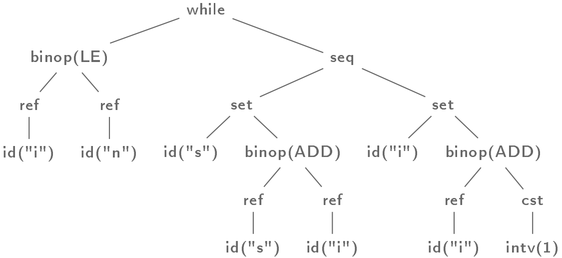
Expressions

$$
\begin{aligned}
E :\ & none \\
\mid\ & cst(C) \\
\mid\ & ref(R) \\
\mid\ & unop(\Omega_1, E) \\
\mid\ & binop(\Omega_2, E, E) \\
\mid\ & ecall(E, E^*) \\
\mid\ & cast(T, E)
\end{aligned}
$$

$$
\begin{aligned}
C :\ & null \\
\mid\ & intv(\mathbb{Z}) \\
\mid\ & floatv(\mathbb{R})
\end{aligned}
$$

$$
\begin{aligned}
\Omega_1 &= \{\mathrm{NEG}, \mathrm{NOT}, \mathrm{INV}\} \\
\Omega_2 &= \{\ \mathrm{ADD}, \mathrm{SUB}, \mathrm{MUL}, \mathrm{DIV}, \mathrm{MOD}, \\
&\qquad \mathrm{LOG\_AND}, \mathrm{LOG\_OR}, \\
&\qquad \mathrm{BIT\_AND}, \mathrm{BIT\_OR}, \mathrm{XOR}, \\
&\qquad \mathrm{SHL}, \mathrm{SHR}, \\
&\qquad \mathrm{EQ}, \mathrm{NE}, \mathrm{LT}, \mathrm{LE}, \mathrm{GT}, \mathrm{GE}\ \}
\end{aligned}
$$

Notation




```
while(
    binop(LE, id("i"), id("n")),
    seq(
        set(id("s"),
            binop(ADD, ref(id("s"), id("i")))),
        set(id("i"),
            binop(ADD, ref(id("i")), (intv(1)))
    )
)
```

Valeurs nulles

Valeurs nulles :
$S \rightarrow \mathit{nop},\ T \rightarrow \mathit{void},\ E \rightarrow \mathit{none},\ C \rightarrow \mathit{null}$

Limiter la complexité des AST :

$$\begin{aligned} \mathit{if}\,(e)\ s_1;\mathit{else}\ s_2 &\Rightarrow \mathit{if}\,(e, s_1, s_2) \\ \mathit{if}\,(e)\ s_1; &\Rightarrow \mathit{if}\,(e, s_1, \mathit{nop}) \end{aligned}$$

$$\begin{aligned} \mathit{return}\ e; &\Rightarrow \mathit{return}(e) \\ \mathit{return}; &\Rightarrow \mathit{return}(\mathit{null}) \end{aligned}$$

Séquence

Definition (Currification)

La curryfication est la transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments.

$$\{s_1, s_2, s_3, ..., s_n\}$$
$$\Updownarrow$$
$$seq(s_1, seq(s_2, seq(s_3, ..., seq(s_{n-1}, s_n)...)))$$
$$\Updownarrow$$
$$seq(...(seq(seq(s_1, s_2), s_3), ...), s_n)$$

Grammaire :

$$\begin{array}{lr} stmts : stmt & \{\$1\} \\ \quad\mid stmts\ stmt & \{SEQ(\$1, \$2)\} \end{array}$$

Suppression de redondance

Pas d'instruction **for** :

$$\textbf{for}(\mathit{init}\,;\ \mathit{cond}\,;\ \mathit{inc})\quad \mathit{stmt} \quad\Leftrightarrow\quad \mathit{init}\ \ \textbf{while}(\mathit{cond})\ \{\ \mathit{stmt}\ \ \mathit{inc}\ \}$$

Expression avec effet de bord :

$$\mathrm{i}{+}{+} \quad\Leftrightarrow\quad \mathrm{i} = \mathrm{i} + 1$$

$$\mathrm{i}{-}{-} \quad\Leftrightarrow\quad \mathrm{i} = \mathrm{i} - 1$$

$$x = e_1\ ?\ e_2 : e_3\,; \quad\Leftrightarrow\quad \mathrm{if}(e_1)\ \ x = e_2\,;\ \ \mathrm{else}\ \ x = e_3\,;$$

Construction

Définition :

```
type stmt =
  | NOP
  ...
  | IF of expr * stmt *stmt
  ...
```

Construction :

$$
\begin{aligned}
\mathit{stmt} : & \ \ldots \\
& \mid \mathit{IF\ LPAR\ expr\ RPAR\ stmt} \\
& \mid \qquad \{\mathit{IF}(\$3, \$5, \mathit{NOP})\} \\
& \mid \mathit{IF\ LPAR\ expr\ RPAR\ stmt\ ELSE\ stmt} \\
& \mid \qquad \{\mathit{IF}(\$3, \$5, \$7)\} \\
& \mid \ldots
\end{aligned}
$$

Plan

Définitions

Definition (Analyseur)

C'est un programme qui (a) prend en entrée un programme et (b) renvoie une valeur calculée sur le programme en entrée.

Definition (Transformeur)

C'est un programme qui (a) prend en entrée un programme et (b) renvoie en sortie un autre programme dans le même langage que le programme en entrée.

Definition (Traducteur)

C'est un programme qui (a) prend en entrée un programme et (b) fournie en sortie un programme dans un autre langage.

Langages

La mise en oeuvre de nos programmes vont utiliser jusqu'à 3 langages :

1. le langage de programmation de l'analyseur / transformeur / traducteur
2. le langage du programme en entrée (*source*)
3. le langage du programme en sortie (*cible*)

Exemples :

- GCC (traducteur programmé en C, source C, cible assembleur)
- passe d'optimisation haut niveau (transformeur programmé en OCAML, source et cible = AST)
- FramaC (analyseur programmé en C++, source C, résultat booléen)

Fonction sur les AST

$$F : \mathrm{AST} \times \mathcal{D} \mapsto \mathcal{D}$$

Exemple :

- F_{type} : $\mathrm{AST} \mapsto \mathbb{B}$ – vérification de typage
- F_{init} : $\mathrm{AST} \times 2^{ID} \mapsto \mathbb{B}$ – vérification que toute variable locale est initialisée avant utilisation
- F_{return} : $\mathrm{AST} \times T \mapsto \mathbb{B}$ – vérification que tout chemin se termine par un *return* du bon type
- F_{local} : $\mathrm{AST} \times \Gamma \mapsto 2^{ID}$ – ensemble des variables locales
- F_{const} : $\mathrm{AST} \mapsto \mathrm{AST}$ – réduction des calculs constants
- ...

Exemple F_{return} (a)

Explorer des cas :

```
int f(int x) {
  if(x < 0)
    return -x;
  else
    return x;
}
```

```
int f(int x) {
  if(x < 0)
    x = -x;
  return x;
}
```

```
int f(int x) {
  if(x < 0)
    x = -x;
  else
    return x;
}
```

```
int f(int x) {
  int s, i = 0;
  while(i < 10) {
    if(g(i))
      return i;
    i++;
  }
}
```

```
int f(int x) {
  while(x < 5) {
    if(x > 2)
      return x;
    x--;
  }
  return x;
}
```

Exemple F_{return} (a)

Explorer des cas :

```
int f(int x) {
  if(x < 0)
    return -x;
  else
    return x;
}
```

```
int f(int x) {
  if(x < 0)
    x = -x;
  return x;
}
```

```
int f(int x) {
  if(x < 0)
    x = -x;
  else
    return x;
}
```

OK

```
int f(int x) {
  int s, i = 0;
  while(i < 10) {
    if(g(i))
      return i;
    i++;
  }
}
```

```
int f(int x) {
  while(x < 5) {
    if(x > 2)
      return x;
    x--;
  }
  return x;
}
```

Exemple F_{return} (a)

Explorer des cas :

```
int f(int x) {
  if(x < 0)
    return -x;
  else
    return x;
}
```

OK

```
int f(int x) {
  if(x < 0)
    x = -x;
  return x;
}
```

OK

```
int f(int x) {
  if(x < 0)
    x = -x;
  else
    return x;
}
```

```
int f(int x) {
  int s, i = 0;
  while(i < 10) {
    if(g(i))
      return i;
    i++;
  }
}
```

```
int f(int x) {
  while(x < 5) {
    if(x > 2)
      return x;
    x--;
  }
  return x;
}
```

Exemple F_{return} (a)

Explorer des cas :

```
int f(int x) {
  if(x < 0)
    return -x;
  else
    return x;
}
```

OK

```
int f(int x) {
  if(x < 0)
    x = -x;
  return x;
}
```

OK

```
int f(int x) {
  if(x < 0)
    x = -x;
  else
    return x;
}
```

Erreur !

```
int f(int x) {
  int s, i = 0;
  while(i < 10) {
    if(g(i))
      return i;
    i++;
  }
}
```

```
int f(int x) {
  while(x < 5) {
    if(x > 2)
      return x;
    x--;
  }
  return x;
}
```

Exemple F_{return} (a)

Explorer des cas :

```
int f(int x) {
  if(x < 0)
    return -x;
  else
    return x;
}
```

OK

```
int f(int x) {
  if(x < 0)
    x = -x;
  return x;
}
```

OK

```
int f(int x) {
  if(x < 0)
    x = -x;
  else
    return x;
}
```

Erreur !

```
int f(int x) {
  int s, i = 0;
  while(i < 10) {
    if(g(i))
      return i;
    i++;
  }
}
```

Erreur !

```
int f(int x) {
  while(x < 5) {
    if(x > 2)
      return x;
    x--;
  }
  return x;
}
```

Exemple F_{return} (a)

Explorer des cas :

```
int f(int x) {
  if(x < 0)
    return -x;
  else
    return x;
}
```

OK

```
int f(int x) {
  if(x < 0)
    x = -x;
  return x;
}
```

OK

```
int f(int x) {
  if(x < 0)
    x = -x;
  else
    return x;
}
```

Erreur !

```
int f(int x) {
  int s, i = 0;
  while(i < 10) {
    if(g(i))
      return i;
    i++;
  }
}
```

Erreur !

```
int f(int x) {
  while(x < 5) {
    if(x > 2)
      return x;
    x--;
  }
  return x;
}
```

OK !

Exemple F_{return} (b)

Expliciter l'analyse réalisée :

> *F_{return} renvoie vrai si, sur l'instruction passée en paramètre, tous les chemins d'exécution se terminent par un **return**.*

Specifier l'analyse :

$$F_{return} : S \rightarrow \mathbb{B}$$

Décrire de manière exhaustive l'analyse :

$$F_{return}[\![nop]\!] = \perp$$

$$\ldots$$

$$F_{return}[\![seq(s_1, s_2)]\!] = F_{return}[\![s_1]\!] \vee F_{return}[\![s_2]\!]$$

$$\ldots$$

Exemple F_{return} (c)

$$
\begin{aligned}
F_{return}[\![nop]\!] &= \bot \\
F_{return}[\![set(\_,\_)]\!] &= \bot \\
F_{return}[\![seq(s_1, s_2)]\!] &= F_{return}[\![s_1]\!] \vee F_{return}[\![s_2]\!] \\
F_{return}[\![if(\_, s_1, s_2)]\!] &= F_{return}[\![s_1]\!] \wedge F_{return}[\![s_2]\!] \\
F_{return}[\![while(\_, s)]\!] &= \bot \\
F_{return}[\![dowhile(s, \_)]\!] &= F_{return}[\![s]\!] \\
F_{return}[\![call(\_,\_)]\!] &= \bot \\
F_{return}[\![return(\_)]\!] &= \top \\
F_{return}[\![block(s)]\!] &= F_{return}[\![s]\!] \\
F_{return}[\![declare(\_,\_,s)]\!] &= F_{return}[\![s]\!]
\end{aligned}
$$

Exercice 1

Écrire une analyse F_{used} qui calcule l'ensemble des variables utilisées par une expression E :

1. évaluez les exemples ci-dessous
2. explicitez
3. specifiez
4. décrivez l'analyse

Expressions :

a $i + 1$

b $x * x + 2 * x + 5$

c $f(i * 2) + i * j$

Exercice 1

Écrire une analyse F_{used} qui calcule l'ensemble des variables utilisées par une expression E :

1. évaluez les exemples ci-dessous
2. explicitez
3. specifiez
4. décrivez l'analyse

Expressions :

- a $i + 1$
- b $x * x + 2 * x + 5$
- c $f(i * 2) + i * j$

a $\{"i"\}$

Exercice 1

Écrire une analyse F_{used} qui calcule l'ensemble des variables utilisées par une expression E :

1. évaluez les exemples ci-dessous
2. explicitez
3. specifiez
4. décrivez l'analyse

Expressions :

- a $i + 1$
- b $x * x + 2 * x + 5$
- c $f(i * 2) + i * j$

- a $\{"i"\}$
- b $\{"x"\}$

Exercice 1

Écrire une analyse F_{used} qui calcule l'ensemble des variables utilisées par une expression E :

1. évaluez les exemples ci-dessous
2. explicitez
3. specifiez
4. décrivez l'analyse

Expressions :

- a $i + 1$
- b $x * x + 2 * x + 5$
- c $f(i * 2) + i * j$

- a $\{"i"\}$
- b $\{"x"\}$
- c $\{"f", "i", "j"\}$

Exercice 1

Écrire une analyse F_{used} qui calcule l'ensemble des variables utilisées par une expression E :

1. évaluez les exemples ci-dessous
2. explicitez
3. specifiez
4. décrivez l'analyse

Expressions :

- a $i + 1$
- b $x * x + 2 * x + 5$
- c $f(i * 2) + i * j$

- a $\{"i"\}$
- b $\{"x"\}$
- c $\{"f", "i", "j"\}$

$$F_{used} : E \mapsto 2^{ID}$$

Exercice 1

Écrire une analyse F_{used} qui calcule l'ensemble des variables utilisées par une expression E :

1. évaluez les exemples ci-dessous
2. explicitez
3. specifiez
4. décrivez l'analyse

Expressions :

a $i + 1$

b $x * x + 2 * x + 5$

c $f(i * 2) + i * j$

a $\{"i"\}$

b $\{"x"\}$

c $\{"f", "i", "j"\}$

$$F_{used} : E \mapsto 2^{ID}$$

$$\begin{aligned}
F_{used}[\![none]\!] &= \emptyset \\
F_{used}[\![cst(_)]\!] &= \emptyset \\
F_{used}[\![ref(id(i))]\!] &= \{i\} \\
F_{used}[\![unop(_, e)]\!] &= F_{used}[\![e]\!] \\
F_{used}[\![binop(_, e_1, e_2)]\!] &= F_{used}[\![e_1]\!] \cup F_{used}[\![e_2]\!] \\
F_{used}[\![ecall(e_1, es)]\!] &= F_{used}[\![e]\!] \\
&\quad \cup \bigcup_{e \in es} F_{used}[\![e]\!] \\
F_{used}[\![cast(_, e)]\!] &= F_{used}[\![e]\!]
\end{aligned}$$

Exercise 2 (a)

La fonction F_{init} a pour objectif de calculer l'ensemble des variables locales qui sont utilisées avant d'être initialisées. Donnez la sortie de F_{init} pour les exemples suivants :

```c
int f(int n) {
  int i = 0, s = 0;
  while(i <= n) {
    s += n;
    i++;
  }
  return s;
}
```

```c
int f(int n) {
  int s = 0, i;
  while(i <= n) {
    s += n;
    i++;
  }
  return s;
}
```

```c
int f(int x) {
  int y;
  if(x < 0) {
    y = -x;
    x--;
  }
  else
    x++;
  return x + y;
}
```

Exercise 2 (a)

La fonction F_{init} a pour objectif de calculer l'ensemble des variables locales qui sont utilisées avant d'être initialisées. Donnez la sortie de F_{init} pour les exemples suivants :

```
int f(int n) {
  int i = 0, s = 0;
  while(i <= n) {
    s += n;
    i++;
  }
  return s;
}
```

$\emptyset$

```
int f(int n) {
  int s = 0, i;
  while(i <= n) {
    s += n;
    i++;
  }
  return s;
}
```

```
int f(int x) {
  int y;
  if(x < 0) {
    y = -x;
    x--;
  }
  else
    x++;
  return x + y;
}
```

Exercise 2 (a)

La fonction F_{init} a pour objectif de calculer l'ensemble des variables locales qui sont utilisées avant d'être initialisées. Donnez la sortie de F_{init} pour les exemples suivants :

```
int f(int n) {
  int i = 0, s = 0;
  while(i <= n) {
    s += n;
    i++;
  }
  return s;
}
```

$\emptyset$

```
int f(int n) {
  int s = 0, i;
  while(i <= n) {
    s += n;
    i++;
  }
  return s;
}
```

$\{"i"\}$

```
int f(int x) {
  int y;
  if(x < 0) {
    y = -x;
    x--;
  }
  else
    x++;
  return x + y;
}
```

Exercise 2 (a)

La fonction F_{init} a pour objectif de calculer l'ensemble des variables locales qui sont utilisées avant d'être initialisées. Donnez la sortie de F_{init} pour les exemples suivants :

```
int f(int n) {
  int i = 0, s = 0;
  while(i <= n) {
    s += n;
    i++;
  }
  return s;
}
```

$\emptyset$

```
int f(int n) {
  int s = 0, i;
  while(i <= n) {
    s += n;
    i++;
  }
  return s;
}
```

$\{"i"\}$

```
int f(int x) {
  int y;
  if(x < 0) {
    y = -x;
    x--;
  }
  else
    x++;
  return x + y;
}
```

$\{"y"\}$

Exercice 2 (b)

En réalité, on peut définir l'ensemble des variables non-initialisées $V_{\overline{init}}$ pour une expressions e comme la différence entre les variables utilisées dans e et les variables initialisées utilisées jusqu'à atteindre e (V_{init}) :

$$V_{\overline{init}} = F_{used}[\![e]\!] \setminus V_{init}$$

Donnez les différentes valeurs de V_{init} dans le programme de droite.

```
int z;
int f(int n) {
  int x, y, s = 0, i;
  /*                */
  if(n > 0) {
    x = -1;
    y = 1;
  }   /*                          */
  else
    x = z;   /*                    */
  /*                    */
  for(i = 0; i != n; i += x)
    s += y;
  /*                        */
  return s;
}
```

Exercice 2 (b)

En réalité, on peut définir l'ensemble des variables non-initialisées $V_{\overline{init}}$ pour une expressions e comme la différence entre les variables utilisées dans e et les variables initialisées utilisées jusqu'à atteindre e (V_{init}) :

$$V_{\overline{init}} = F_{used}[\![e]\!] \setminus V_{init}$$

Donnez les différentes valeurs de V_{init} dans le programme de droite.

```
int z;
int f(int n) {
  int x, y, s = 0, i;
  /*{"n","s","z"}*/
  if(n > 0) {
    x = -1;
    y = 1;
  }   /*                  */
  else
    x = z;  /*                */
  /*               */
  for(i = 0; i != n; i += x)
    s += y;
  /*                    */
  return s;
}
```

Exercice 2 (b)

En réalité, on peut définir l'ensemble des variables non-initialisées $V_{\overline{init}}$ pour une expressions e comme la différence entre les variables utilisées dans e et les variables initialisées utilisées jusqu'à atteindre e (V_{init}) :

$$V_{\overline{init}} = F_{used}[\![e]\!] \setminus V_{init}$$

Donnez les différentes valeurs de V_{init} dans le programme de droite.

```
int z;
int f(int n) {
  int x, y, s = 0, i;
  /*{"n","s","z"}*/
  if(n > 0) {
    x = -1;
    y = 1;
  }   /*{"n","s","x","y","z"}*/
  else
    x = z;  /*                */
  /*               */
  for(i = 0; i != n; i += x)
    s += y;
  /*                  */
  return s;
}
```

Exercice 2 (b)

En réalité, on peut définir l'ensemble des variables non-initialisées $V_{\overline{init}}$ pour une expressions e comme la différence entre les variables utilisées dans e et les variables initialisées utilisées jusqu'à atteindre e (V_{init}) :

$$V_{\overline{init}} = F_{used}[\![e]\!] \setminus V_{init}$$

Donnez les différentes valeurs de V_{init} dans le programme de droite.

```
int z;
int f(int n) {
  int x, y, s = 0, i;
  /*{"n","s","z"}*/
  if(n > 0) {
    x = -1;
    y = 1;
  }   /*{"n","s","x","y","z"}*/
  else
    x = z;  /*{"n","s","x","z"}*/
  /*                  */
  for(i = 0; i != n; i += x)
    s += y;
  /*                      */
  return s;
}
```

Exercice 2 (b)

En réalité, on peut définir l'ensemble des variables non-initialisées $V_{\overline{init}}$ pour une expressions e comme la différence entre les variables utilisées dans e et les variables initialisées utilisées jusqu'à atteindre e (V_{init}) :

$$V_{\overline{init}} = F_{used}[\![e]\!] \setminus V_{init}$$

Donnez les différentes valeurs de V_{init} dans le programme de droite.

```
int z;
int f(int n) {
  int x, y, s = 0, i;
  /*{"n","s","z"}*/
  if(n > 0) {
    x = -1;
    y = 1;
  }   /*{"n","s","x","y","z"}*/
  else
    x = z;  /*{"n","s","x","z"}*/
  /*{"n","s","x","z"}*/
  for(i = 0; i != n; i += x)
    s += y;
  /*                    */
  return s;
}
```

Exercice 2 (b)

En réalité, on peut définir l'ensemble des variables non-initialisées $V_{\overline{init}}$ pour une expressions e comme la différence entre les variables utilisées dans e et les variables initialisées utilisées jusqu'à atteindre e (V_{init}) :

$$V_{\overline{init}} = F_{used}[\![e]\!] \setminus V_{init}$$

Donnez les différentes valeurs de V_{init} dans le programme de droite.

```
int z;
int f(int n) {
  int x, y, s = 0, i;
  /*{"n","s","z"}*/
  if(n > 0) {
    x = -1;
    y = 1;
  }   /*{"n","s","x","y","z"}*/
  else
    x = z;  /*{"n","s","x","z"}*/
  /*{"n","s","x","z"}*/
  for(i = 0; i != n; i += x)
    s += y;
  /*{"i","n","s","x","z"}*/
  return s;
}
```

Exercice 2 (c)

Nous allons d'abord écrire la fonction F_{init} qui calcule l'ensemble des variables initialisées pour chaque point du programme. Comme $V_{init} \subseteq 2^{ID}$, on a la spécification :

$$F_{init} : S \times 2^{ID} \mapsto 2^{ID}$$

Donnez la description de F_{init} en prenant comme exemple :

$$F_{init}[\![nop]\!]\ V = V$$

Exercice 2 (c)

Nous allons d'abord écrire la fonction F_{init} qui calcule l'ensemble des variables initialisées pour chaque point du programme. Comme $V_{init} \subseteq 2^{ID}$, on a la spécification :

$$F_{init} : S \times 2^{ID} \mapsto 2^{ID}$$

Donnez la description de F_{init} en prenant comme exemple :

$$F_{init}[\![nop]\!]\ V = V$$

$$\begin{aligned}
F_{init}[\![set(ref(id(i)), _)]\!]\ V &= V \cup \{i\} \\
F_{init}[\![seq(s_1, s_2)]\!]\ V &= F_{init}[\![s_2]\!]\ (F_{init}[\![s_1]\!]\ V) \\
F_{init}[\![if(_, s_1, s_2)]\!]\ V &= (F_{init}[\![s_1]\!]\ V) \cap (F_{init}[\![s_2]\!]\ V) \\
F_{init}[\![while(_, s)]\!]\ V &= V \\
F_{init}[\![dowhile(s, _)]\!]\ V &= F_{init}[\![s]\!]\ V \\
F_{init}[\![call(_, _)]\!]\ V &= V \\
F_{init}[\![return(_)]\!]\ V &= V \\
F_{init}[\![block(s)]\!]\ V &= F_{init}[\![s]\!]\ V \\
F_{init}[\![declare(_, _, s)]\!]\ V &= F_{init}[\![s]\!]\ V
\end{aligned}$$

Exercice 2 (d)

On dispose de la fonction $check_{init}$ qui prend en paramètre un ensemble d'expressions et la liste des variables initialisées et lève éventuellement une erreur de non-initialisation. Re-écrire F_{init} pour détecter les variables non-initialisées.

$$check_{init}\ E\ V = if\ (\bigcup_{e \in E} F_{used}[\![e]\!]) \setminus V \neq \emptyset\ then\ error$$

Exercice 2 (d)

On dispose de la fonction $check_{init}$ qui prend en paramètre un ensemble d'expressions et la liste des variables initialisées et lève éventuellement une erreur de non-initialisation. Re-écrire F_{init} pour détecter les variables non-initialisées.

$$check_{init}\ E\ V = if\ (\bigcup_{e \in E} F_{used}[\![e]\!]) \setminus V \neq \emptyset\ then\ error$$

$$\begin{aligned}
F_{init}[\![set(ref(id(i)), e)]\!]\ V &= check_{init}\ \{e\}\ V; V \cup \{i\} \\
F_{init}[\![seq(s_1, s_2)]\!]\ V &= F_{init}[\![s_2]\!]\ (F_{init}[\![s_1]\!]\ V) \\
F_{init}[\![if(e, s_1, s_2)]\!]\ V &= check_{init}\ \{e\}\ V; (F_{init}[\![s_1]\!]\ V) \cap (F_{init}[\![s_2]\!]\ V) \\
F_{init}[\![while(e, s)]\!]\ V &= check_{init}\ \{e\}\ V; V \cap (F_{init}[\![s]\!]\ V) \\
F_{init}[\![dowhile(s, e)]\!]\ V &= check_{init}\ \{e\}\ V; V \\
F_{init}[\![call(e, es)]\!]\ V &= check_{init}\ (\{e\} \cup es)\ V; V \\
F_{init}[\![return(e)]\!]\ V &= check_{init}\ \{e\}\ V; V \\
F_{init}[\![block(s)]\!]\ V &= F_{init}[\![s]\!]\ V \\
F_{init}[\![declare(_, _, s)]\!]\ V &= F_{init}[\![s]\!]\ V
\end{aligned}$$

Bilan partiel

Faire une analyses :

1. Identifier un domaine de calcul.
2. Ecrire le calcul pour chaque constructeur d'AST

Problème : jonction trouvée dans *if*, *while*, *dowhile* :
2 chemins d'exécution possible se séparent et se rejoignent.

2 manières de traiter la jonction de chemin :

- la propriété doit être vraie sur tous chemins (*MUST*) – opérateur de jonction restrictif ($\sqcap$)
- la propriété doit être vraie sur au moins un chemin (*MAY*) – opérateur de jonction extensif ($\sqcup$)

Exemples de domaines : $\mathbb{B}$ ($\wedge$, $\vee$), ensembles ($\cap$, $\cup$), etc

Plan

Vérification des types

$$\begin{array}{c} \text{AST non-typé} \\ \downarrow \\ \boxed{\text{Vérification de types}} \longrightarrow \text{erreur} \\ \downarrow \\ \text{AST typé} \end{array}$$

Traducteur :

$$F_T : \text{AST} \times \Gamma \mapsto \text{AST}_T$$

Avec Γ l'ensemble des environnements :

- $\forall \gamma \in \Gamma, \gamma : ID \rightarrow T$
- $\gamma_\perp = \lambda i.void$ – environnement vide
- $\gamma[i]$ – type de la variable i
- $\gamma[i] = void \Leftrightarrow i$ n'est pas déclaré
- $\gamma[i \leftarrow x] = \lambda i'.if\ \ i = i'\ \ then\ x\ \ else\ \gamma[i]$ – affectation

AST typé

$$
\begin{aligned}
S_T :\ & nop \\
\mid\ & set(R, E_T : T) \\
\mid\ & seq(S_T, S_T) \\
\mid\ & if(E_T : T, S_T, S_T) \\
\mid\ & while(E_T : T, S_T) \\
\mid\ & dowhile(S_T, E_T : T) \\
\mid\ & call(E_T : T, (E_T : T)^*) \\
\mid\ & return(E_T : T) \\
\mid\ & block(S_T) \\
\mid\ & declare(ID, T, S_T)
\end{aligned}
$$

$$
\begin{aligned}
E_T :\ & none \\
\mid\ & cst(C) \\
\mid\ & ref(R) \\
\mid\ & unop(\Omega_1, E_T : T) \\
\mid\ & binop(\Omega_2, E_T : T, E_T : T) \\
\mid\ & ecall(E_T : T, (E_T : T)^*) \\
\mid\ & cast(T, E_T : T)
\end{aligned}
$$

Exemple

$$
\begin{array}{l}
block(seq(\\
\quad declare("i", int, \\
\quad declare("s", int, \\
\quad set("i", cst(intv(0))), seq(\\
\quad set("s", cst(intv(0))), seq(\\
\quad while(binop(LE, ref(id("i")), ref(id("n")))), seq(\\
\quad\quad set(id("s"), binop(ADD, ref(id("s")), ref(id("i")))), \\
\quad\quad set(id("i"), binop(ADD, ref(id("i")), cst(intv(1))))), \\
\quad return(ref(id("s))))))))))
\end{array}
$$

Exemple

```
block(seq(
    declare("i", int,
    declare("s", int,
    set("i", cst(intv(0))), seq(
    set("s", cst(intv(0))), seq(
    while(binop(LE, ref(id("i")), ref(id("n"))), seq(
        set(id("s"), binop(ADD, ref(id("s")), ref(id("i")))),
        set(id("i"), binop(ADD, ref(id("i")), cst(intv(1))))),
    return(ref(id("s)))))))))
```

$\Downarrow$

```
block(seq(
    declare("i", int,
    declare("s", int,
    set("i", cst(intv(0)): int), seq(
    set("s", cst(intv(0)): int), seq(
    while(binop(LE, ref(id("i")): int, ref(id("n")): int): int, seq(
        set(id("s"), binop(ADD, ref(id("s")): int, ref(id("i")): int): int),
        set(id("i"), binop(ADD, ref(id("i")): int, cst(intv(1)): int): int)),
    return(ref(id("s)): int))))))))
```

Gestion de l'environnement

Fonction f :

- paramètres $P \subseteq 2^{ID \times T}$
- variables globales $G \subseteq 2^{ID \times T}$
- corps $s \in S$

Gestion de l'environnement

Fonction f :

- paramètres $P \subseteq 2^{ID \times T}$
- variables globales $G \subseteq 2^{ID \times T}$
- corps $s \in S$

$$\gamma_f = \lambda i. \begin{cases} \tau & \mathit{if}\ (i, \tau) \in P \cup G \\ \mathit{void} & \mathit{else} \end{cases}$$

Gestion de l'environnement

Fonction f :

- paramètres $P \subseteq 2^{ID \times T}$
- variables globales $G \subseteq 2^{ID \times T}$
- corps $s \in S$

$$\gamma_f = \lambda i. \begin{cases} \tau & \mathit{if}\ (i, \tau) \in P \cup G \\ \mathit{void} & \mathit{else} \end{cases}$$

$$\mathit{let}\ s' = F_t[\![s]\!]\ \gamma_f$$

Gestion de l'environnement

Fonction f :

- paramètres $P \subseteq 2^{ID \times T}$
- variables globales $G \subseteq 2^{ID \times T}$
- corps $s \in S$

$$\gamma_f = \lambda i. \begin{cases} \tau & \text{if } (i, \tau) \in P \cup G \\ void & else \end{cases}$$

$$let\ s' = F_t[\![s]\!]\ \gamma_f$$

$$\begin{aligned} F_t[\![nop]\!]\ \gamma &= nop \\ F_t[\![declare(i, \tau, s)]\!]\ \gamma &= declare(i, t, F_t[\![s]\!]\ \gamma[i \to \tau]) \\ F_t[\![block(s)]\!]\ \gamma &= block(F_t[\![s]\!]\ \gamma) \\ F_t[\![seq(s_1, s_2)]\!]\ \gamma &= seq(F_t[\![s_1]\!]\ \gamma, F_t[\![s_2]\!]\ \gamma) \end{aligned}$$

Affectation

$$F_t[\![set(i,e)]\!]\ \gamma = let\ e' : \tau' = F_t[\![e]\!]\ \gamma\ in$$

$$\begin{cases} (set(i, e' : \tau'), \gamma) & if\, \gamma[i] = \tau' \\ (set(i, cast(\tau, e' : \tau') : \tau), \gamma) & if\, \gamma[i] = \tau \wedge acv(\tau, \tau') \\ erreur & sinon \end{cases}$$

Affectation

$$F_t[\![set(i,e)]\!]\ \gamma = let\ e' : \tau' = F_t[\![e]\!]\ \gamma\ in$$
$$\begin{cases} (set(i, e' : \tau'), \gamma) & if\, \gamma[i] = \tau' \\ (set(i, cast(\tau, e' : \tau') : \tau), \gamma) & if\, \gamma[i] = \tau \wedge acv(\tau, \tau') \\ erreur & sinon \end{cases}$$

2 erreurs possibles :

- variable non déclarée $\Leftrightarrow \gamma[i] = void$
- conversion impossible $\Leftrightarrow acv(\tau', \tau) = \bot$

Affectation

$$F_t[\![set(i,e)]\!]\ \gamma = let\ e' : \tau' = F_t[\![e]\!]\ \gamma\ in$$

$$\begin{cases} (set(i, e' : \tau'), \gamma) & if\, \gamma[i] = \tau' \\ (set(i, cast(\tau, e' : \tau') : \tau), \gamma) & if\, \gamma[i] = \tau \wedge acv(\tau, \tau') \\ erreur & sinon \end{cases}$$

2 erreurs possibles :

- variable non déclarée $\Leftrightarrow \gamma[i] = void$
- conversion impossible $\Leftrightarrow acv(\tau', \tau) = \perp$

$$\forall \tau, \tau' \in T, acv(\tau', \tau) = (\tau', \tau) \in \{$$
$$(int, char), (char, int),$$
$$(float, int), (int, float),$$
$$(float, char), (char, float)$$
$$\}$$

Note Il est facile d'ajouter un nouveau type.

Opérateur unaire

Facile : résultat du même type ou alors conversion vers entier.

$$F_t[\![unop(\omega_1, e : \tau)]\!]\ \gamma = let\ e' : \tau' = F_t[\![e]\!]\ \gamma\ in$$
$$\begin{cases} unop(\omega_1, e : \tau') : int & if\ \tau' = int \vee\ \tau = float \\ unop(\omega_1, cast(int, e' : \tau') : int) : int & if\ acv(int, \tau') \\ erreur & sinon \end{cases}$$

Remarque même procédure

- sous-instructions ou sous-expressions sont typées
- vérification que le type appartient à un modèle accepté
- insertion de conversion automatique
- sinon une erreur est levée !

Opérateur unaire (version 2)

Le typage des sous-instructions et des sous-expressions est implicite

$$\frac{\gamma \vdash unop(\omega, e : int)}{\gamma \vdash unop(\omega, e : int) : int}$$

$$\frac{\gamma \vdash unop(\omega, e : char)}{\gamma \vdash unop(\omega, cast(int, e : char) : int) : int}$$

Peut-être utilisé pour les instructions :

$$\frac{\gamma \vdash set(i, e : \tau) \wedge \gamma[i] = \tau}{\gamma \vdash set(i, e : \tau)}$$

$$\frac{\gamma \vdash set(i, e : int) \wedge \gamma[i] = char}{\gamma \vdash set(i, cast(char, e : int))}$$

$$\frac{\gamma \vdash set(i, e : char) \wedge \gamma[i] = INT}{\gamma \vdash set(i, cast(int, e : char))}$$

$$\ldots$$

Règle : liste exhaustive des conversions possibles *Rightarrow* facilité de vérification

Opérateur binaire

$$\frac{\gamma \vdash binop(\omega, e : int, e' : int) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, e : int, e' : int) : INT}$$

$$\frac{\gamma \vdash binop(\omega, e : int, e' : char) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, e : int, cast(char, e' : char) : int) : int}$$

$$\frac{\gamma \vdash binop(\omega, e : char, e' : int) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, cast(int, e : char) : int, e' : int) : int}$$

$$\frac{\gamma \vdash binop(\omega, e : char, e' : char) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, cast(int, e : char) : int, cast(char, e' : int) : int) : int}$$

Opérateur binaire

$$\frac{\gamma \vdash binop(\omega, e : int, e' : int) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, e : int, e' : int) : INT}$$

$$\frac{\gamma \vdash binop(\omega, e : int, e' : char) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, e : int, cast(char, e' : char) : int) : int}$$

$$\frac{\gamma \vdash binop(\omega, e : char, e' : int) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, cast(int, e : char) : int, e' : int) : int}$$

$$\frac{\gamma \vdash binop(\omega, e : char, e' : char) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, cast(int, e : char) : int, cast(char, e' : int) : int) : int}$$

$$\frac{\gamma \vdash binop(\omega, e : float, e' : float) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, e : float, e' : float) : float}$$

$$\frac{\gamma \vdash binop(\omega, e : float, e' : int) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, e : float, cast(float, e' : int) : float) : float}$$

$$\frac{\gamma \vdash binop(\omega, e : float, e' : char) \wedge \omega \in \{ADD, SUB, MUL, DIV\}}{\gamma \vdash binop(\omega, e : float, cast(float, e' : char) : float) : float}$$

$\ldots$

Exercice 3

Donnez les règles de typage des constructions suivantes :

- $cst(c)$
- $ref(id(i))$
- $while(e, s)$
- $if(e, s_1, s_2)$

Exercice 3

Donnez les règles de typage des constructions suivantes :

- $cst(c)$
- $ref(id(i))$
- $while(e, s)$
- $if(e, s_1, s_2)$

$$\frac{\gamma \vdash cst(intv(n))}{\gamma \vdash: cst(intv(n)) : int}$$

$$\frac{\gamma \vdash cst(floatv(x))}{\gamma \vdash: cst(floatv(x)) : float}$$

Exercice 3

Donnez les règles de typage des constructions suivantes :

- $cst(c)$
- $ref(id(i))$
- $while(e, s)$
- $if(e, s_1, s_2)$

$$\frac{\gamma \vdash cst(intv(n))}{\gamma \vdash: cst(intv(n)) : int}$$

$$\frac{\gamma \vdash cst(floatv(x))}{\gamma \vdash: cst(floatv(x)) : float}$$

$$\frac{\gamma \vdash id(ref(i)) \wedge \gamma[i] \neq void}{\gamma \vdash id(ref(i)) : \gamma[i]}$$

Exercice 3

Donnez les règles de typage des constructions suivantes :

- $cst(c)$
- $ref(id(i))$
- $while(e, s)$
- $if(e, s_1, s_2)$

$$\frac{\gamma \vdash cst(intv(n))}{\gamma \vdash : cst(intv(n)) : int}$$

$$\frac{\gamma \vdash cst(floatv(x))}{\gamma \vdash : cst(floatv(x)) : float}$$

$$\frac{\gamma \vdash id(ref(i)) \wedge \gamma[i] \neq void}{\gamma \vdash id(ref(i)) : \gamma[i]}$$

$$\frac{\gamma \vdash while(e : int, s)}{\gamma \vdash while(e : int, s)}$$

$$\frac{\gamma \vdash while(e : char, s)}{\gamma \vdash while(cast(int, e : char) : int, s)}$$

Exercice 3

Donnez les règles de typage des constructions suivantes :

- $cst(c)$
- $ref(id(i))$
- $while(e, s)$
- $if(e, s_1, s_2)$

$$\frac{\gamma \vdash cst(intv(n))}{\gamma \vdash : cst(intv(n)) : int}$$

$$\frac{\gamma \vdash cst(floatv(x))}{\gamma \vdash : cst(floatv(x)) : float}$$

$$\frac{\gamma \vdash id(ref(i)) \wedge \gamma[i] \neq void}{\gamma \vdash id(ref(i)) : \gamma[i]}$$

$$\frac{\gamma \vdash while(e : int, s)}{\gamma \vdash while(e : int, s)}$$

$$\frac{\gamma \vdash while(e : char, s)}{\gamma \vdash while(cast(int, e : char) : int, s)}$$

$$\frac{\gamma \vdash if(e : int, s_1, s_2)}{\gamma \vdash if(e : int, s_1, s_2)}$$

$$\frac{\gamma \vdash if(e : char, s_1, s_2)}{\gamma \vdash if(cast(int, e : char) : int, s_1, s_2)}$$

Exercice 4

Donnez les règles de typage des constructions suivantes :

- $declare(i, \tau, s)$
- $return(e)$
- $call(e, [e_1, e_2, ..., e_n])$

Exercice 4

Donnez les règles de typage des constructions suivantes :

- $declare(i, \tau, s)$
- $return(e)$
- $call(e, [e_1, e_2, ..., e_n])$

$$\frac{\gamma \vdash declare(i, \tau, s) \wedge \gamma[i \rightarrow \tau] \vdash s}{\gamma \vdash declare(i, \tau, s)}$$

Exercice 4

Donnez les règles de typage des constructions suivantes :

- $declare(i, \tau, s)$
- $return(e)$
- $call(e, [e_1, e_2, ..., e_n])$

$$\frac{\gamma \vdash declare(i, \tau, s) \wedge \gamma[i \rightarrow \tau] \vdash s}{\gamma \vdash declare(i, \tau, s)}$$

$$\frac{\gamma \vdash return(e : \tau) \wedge \tau = \gamma["\$return"]}{\gamma \vdash return(e : \tau)}$$

"$return" spécial positionné dans γ_f

Exercice 4

Donnez les règles de typage des constructions suivantes :

- $declare(i, \tau, s)$
- $return(e)$
- $call(e, [e_1, e_2, ..., e_n])$

$$\frac{\gamma \vdash declare(i, \tau, s) \wedge \gamma[i \rightarrow \tau] \vdash s}{\gamma \vdash declare(i, \tau, s)}$$

$$\frac{\gamma \vdash return(e : \tau) \wedge \tau = \gamma[''\$return'']}{\gamma \vdash return(e : \tau)}$$

"\$return" spécial positionné dans γ_f

$$\frac{\begin{array}{l} \gamma \vdash call(e : fun(void, [\tau_1, ..., \tau_n]), \\ \qquad e_1 : \tau'_1, ..., e_n : \tau'_n) \\ \forall i \in [\![1, n]\!], (\tau_i = \tau'_i \wedge x_i = e_i) \vee \\ \qquad (acv(\tau_i, \tau'_i) \wedge x_i = cast(\tau_i, e_i : \tau'_i) : \tau_i) \end{array}}{\begin{array}{l} \gamma \vdash call(e : fun(void, [\tau_1, ..., \tau_n]), \\ \qquad x_1 : \tau_1, ..., x_n : \tau_n) \end{array}}$$

Exercice 5

Nous introduisons le type pointeur :

$$T : \ldots \mid ptr(T)$$

Et des opérateurs sur les pointeurs :

$$R : \ldots \\ \mid at(E)$$

Opérateur "*".

$$E : \ldots \\ \mid addr(R)$$

Opérateur "&".

Donnez le typage de :

1. $at(e)$ et $addr(r)$
2. addition / soustraction – pointeur / entier
3. soustraction entre 2 pointeurs

Exercice 5 (correction)

Exercice 5 (correction)

$$\frac{\gamma \vdash at(e : ptr(\tau))}{\gamma \vdash at(e : ptr(\tau)) : \tau}$$

$$\frac{\gamma \vdash addr(r : \tau)}{\gamma \vdash addr(r : \tau) : ptr(\tau)}$$

Exercice 5 (correction)

$$\frac{\gamma \vdash at(e : ptr(\tau))}{\gamma \vdash at(e : ptr(\tau)) : \tau}$$

$$\frac{\gamma \vdash addr(r : \tau)}{\gamma \vdash addr(r : \tau) : ptr(\tau)}$$

$$\frac{\gamma \vdash binop(\omega, e_1 : ptr(\tau), e_2 : int) \wedge \omega \in \{ADD, SUB\}}{\gamma \vdash binop(\omega, e_1 : ptr(\tau), e_2 : int) : ptr(\tau)}$$

$$\frac{\gamma \vdash binop(ADD, e_1 : int, e_2 : ptr(\tau))}{\gamma \vdash binop(\omega, e_2 : ptr(\tau), e_1 : int) : ptr(\tau)}$$

$$\frac{\gamma \vdash binop(\omega, e_1 : ptr(\tau), e_2 : char) \wedge \omega \in \{ADD, SUB\}}{\gamma \vdash binop(\omega, e_1 : ptr(\tau), cast(int, e_2 : char) : int) : ptr(\tau)}$$

$$\frac{\gamma \vdash binop(ADD, e_1 : char, e_2 : ptr(\tau))}{\gamma \vdash binop(\omega, e_2 : ptr(\tau), cast(int, e_1 : char) : int) : ptr(\tau)}$$

Exercice 5 (correction)

$$\frac{\gamma \vdash at(e : ptr(\tau))}{\gamma \vdash at(e : ptr(\tau)) : \tau}$$

$$\frac{\gamma \vdash addr(r : \tau)}{\gamma \vdash addr(r : \tau) : ptr(\tau)}$$

$$\frac{\gamma \vdash binop(\omega, e_1 : ptr(\tau), e_2 : int) \wedge \omega \in \{ADD, SUB\}}{\gamma \vdash binop(\omega, e_1 : ptr(\tau), e_2 : int) : ptr(\tau)}$$

$$\frac{\gamma \vdash binop(ADD, e_1 : int, e_2 : ptr(\tau))}{\gamma \vdash binop(\omega, e_2 : ptr(\tau), e_1 : int) : ptr(\tau)}$$

$$\frac{\gamma \vdash binop(\omega, e_1 : ptr(\tau), e_2 : char) \wedge \omega \in \{ADD, SUB\}}{\gamma \vdash binop(\omega, e_1 : ptr(\tau), cast(int, e_2 : char) : int) : ptr(\tau)}$$

$$\frac{\gamma \vdash binop(ADD, e_1 : char, e_2 : ptr(\tau))}{\gamma \vdash binop(\omega, e_2 : ptr(\tau), cast(int, e_1 : char) : int) : ptr(\tau)}$$

$$\frac{\gamma \vdash binop(SUB, e_1 : ptr(\tau), e_2 : ptr(\tau))}{\gamma \vdash binop(SUB, e_1 : ptr(\tau), e_2 : ptr(\tau)) : int}$$

Exercice 6 (hors cours)

Nous introduisons maintenant le type tableau :

$$T : ... \mid array(T, \mathbb{N})$$

Et l'opérateur d'accès au tableau :

$$R : ... \mid item(tab : E, index : E)$$

1. Donnez le typage de $item(e_1, e_2)$.
2. En considérant que *item* peut être aussi utilisé avec des pointeurs, donnez le typage correspondant.
3. On peut affecter un tableau à un pointeur : donnez le typage correspondant avec $set(r, e)$.
4. A-t-on vraiment besoin du constructeur *item* ? Proposez une représentation en utilisant les autres constructeurs.

Plan

Conclusion

- l'analyse sémantique permet de vérifier toutes les règles non sémantiques du langage
- l'objectif est que l'AST satisfasse complètement les conditions de la traduction
- après cette phase, le programme est considéré juste d'un point de vue *statique*
- le back-end peut être mis en œuvre sereinement