Compilateur

Raphael Javaux

Plan

- Langage source
- Compilateur
- Démonstration
- Implémentation
  - Parseur
  - Générateur de code

Langage source

- Syntaxe « C » ;
- `int`, `bool` et tableaux ;
- `if-else` et `while` ;
- Inférence du type des variables initialisées ;
- Commentaires ;
- Pas d'I/O.

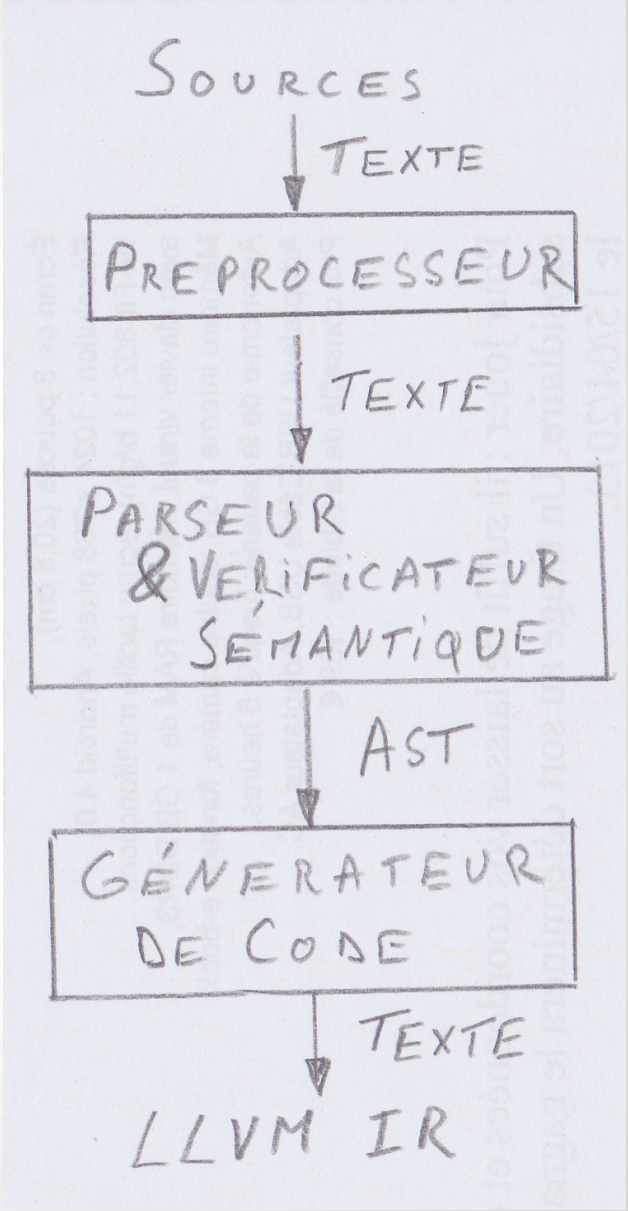
```
quicksort(int[] v, int left, int right)
{
    if (left < right) {
        auto i = left + 1;
        auto last = left;
        while (i <= right) {
            if (v[i] < v[left]) {
                last = last + 1;
                swap(v, last, i);
            }
            i = i+1;
        }

        swap(v, left, last);
        quicksort(v, left, last - 1);
        quicksort(v, last + 1, right);
    }
}

swap(int[] v, int i, int j)
{
    auto tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;
}
```

Compilateur

- Implémenté en Haskell :
  - Langage fonctionnel pur ;
  - Typage statique fort, inféré ;
- Parsec ;
- Trois phases :
  - Précompilation ;
  - Parsing et vérification sémantique ;
  - Génération du code LLVM.

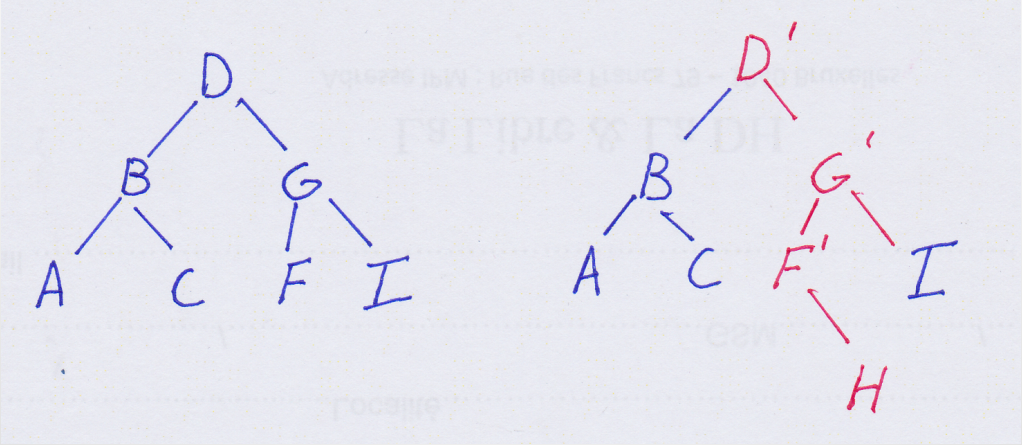


	blank	comment	code
Définition de l'arbre syntaxique	30	21	81
Préprocesseur	2	3	4
Parseur et vérificateur sémantique	74	72	344
Générateur de code	66	51	240
Autre	2	0	12
SUM	174	147	681

Démonstration

Implémentation du parseur

- Parsec :
  - Parseur primitifs simples combinables ;
  - `string`, `char`, `spaces`, alternative... ;
  - Gestion des erreurs et du backtracking ;
- Table de symboles immuable :
  - Nouvelle scope (copie) : O(1) ;
  - Nouvelle variable (insertion) : O(log n).



Définition du type d'une variable

`const int, auto, const auto, bool, int ...`

```haskell
-- Combinateur retournant la valeur CQualConst s'il est arrivé à parser la
-- chaine "const", CQualFree sinon.
typeQual =     (string "const" >> return CQualConst)
           <|> return CQualFree

-- Combinateur parsant les chaines "int" ou "bool".
typeSpec = string "int" <|> string "bool"

-- Combinateur parsant le type d'une variable.
-- Retourne le qualificateur et le type éventuel (si non-'auto').
varType = do
    qual <- typeQual
    spaces                          -- Ignore les espaces après la qualificateur.
    (    (do t <- typeSpec
             return (qual, Just t))
     <|> (string "auto" >> return (qual, Nothing)))
```

Implémentation du générateur de code

- Génère du LLVM IR sous forme de texte ;
  - N'utilise pas l'API d'LLVM ;
- Monades gérant l'état du générateur :
  - Etat du flux de caractères et des compteurs ;
  - Passage implicite de l'état du générateur de code ;

```haskell
-- Génère une instruction while composée d'une garde nommé 'guard' et d'un bloc
-- d'instruction nommé 'stmts'.
cStmt (CWhile guard stmts) = do
    guardLabel <- newLabel -- newLabel alloue un nouveau label.
    loopLabel  <- newLabel -- Cette fonction incrémente le compteur des
    endLabel   <- newLabel -- identifiants des labels de la monade State et
                           -- retourne le nom du nouveau label correspondant.
    branch guardLabel      -- Génère les instructions pour brancher sur le label
                           -- de la garde de l'instruction while sur le flux de
                           -- caractères de la monade Writer.

    tellLabel guardLabel   -- Ajoute sur le flux de caractères l'instruction
                           -- LLVM indiquant le début d'un nouveau bloc nommé
                           -- avec le label précédemment alloué.
    cond <- cExpr guard    -- Génère le code la garde et retourne le nom du
                           -- registre contenant le résultat.
    branchCond cond loopLabel endLabel -- Branche en fonction de la valeur du
                                       -- registre retourné par la génération du
                                       -- code de la condition.

    tellLabel loopLabel
    cCompoundStmt stmts    -- Génère (récusivement) le code des instructions du
                           -- bloc de code de l'instruction while.
    branch guardLabel

    tellLabel endLabel
```

```haskell
-- La fonction de génération accepte ici deux arguments supplémentaires pour le
-- flux de caractère et les compteurs.
cStmt stream0 counters0 (CWhile guard stmts) = do
    (guardLabel, stream1, counters1) <- newLabel stream0 counters0
    (loopLabel, stream2, counters2)  <- newLabel stream1 counters1
    (endLabel, stream3, counters3)   <- newLabel stream2 counters2

    (stream4, counters4) <- branch stream3 counters3 guardLabel

    (stream5, counters5) <- tellLabel stream4 counters4 guardLabel
    (cond, stream6, counters6) <- cExpr stream5 counters5 guard
    (stream7, counters7) <- branchCond stream6 counters6 cond loopLabel endLabel

    (stream8, counters8) <- tellLabel stream7 counters7 loopLabel
    (stream9, counters9) <- cCompoundStmt stream8 counters8 stmts
    (stream10, counters10) <- branch stream9 counters9 guardLabel

    (stream11, counters11) <- tellLabel stream10 counters10 endLabel

    -- Retourne les nouveaux états des flux de caractères et des compteurs.
    return (stream11, counters11)
```

Questions