UNIVERSITÉ DE ROUEN

Compilateur LLVM

Langage jouet Kawa

Module d'analyse syntaxique 0.1

6 avril 2015

Auteur(s): Kheireddine Berkane, Abdellatif Ahouate

Table des matières

1 Analyse lexicale

1.1 Objectif de l'analyse lexicale

le principale objectif de cette analyse est de reconnaitre des unités lexicales (tokens). Chaque token est défini par une expression rationnelle, les entités reconnues lors de cette phase serviront par la suite comme entrée de la phase d'analyse syntaxique.

1.2 Lexer

```
/* Fonction qui compte les lignes et les colonnes pour préciser la position d'une erreur */
void count() {
  int i;
  column += strlen(yytext);
  if(strcmp(yytext, "\t")==0)
   column +=3;
  for (i = 0; yytext[i] != '\0'; i++) {
    if (yytext[i] == '\n'){
      column = 1;
      lineno++;
    }
  }
}

/*--Partie qui définit les entiers, les réels et les identificateurs ------*/
INT [0-9]+
FLOAT [0-9]+(\.[0-9]+)?(e[0-9]+)?
ID [_]?[a-zA-Z][a-zA-Z0-9]*

/*--Partie qui permet de reconnaitre les entiers et les réels pour les renvoyer au parser---*/
{INT} {count(); yylval.vint=atof(yytext); return(ENTIER); }
{FLOAT} {count(); yylval.vint=atof(yytext); return(REEL); }

/*--Partie qui permet de reconnaitre  ----*/
/*--les mots clés du langage source (kawa) pour les renvoyer au parser----*/
"string"                {count(); return (TSTRING);}
"short"                 {count(); return (TSHORT);}
"int"                   {count(); return (TINT);}
"long"                  {count(); return (TLONG);}
"float"                 {count(); return (TFLOAT);}
"double"                {count(); return (TDOUBLE);}
"byte"                  {count(); return (TBYTE);}
"char"                  {count(); return (TCHAR);}
"boolean"               {count(); return (TBOOLEAN);}
"void"                  {count(); return (TVOID);}
"const"                 {count(); /*return (TCONST);*/}
"enum"                  {count(); /*return (TENUM);*/}
```

```
"value"                 {count(); return (TVALUE);}
"import"                {count(); return (TIMPORT);}
"public"                {count(); return (TPUBLIC);}
"private"               {count(); return (TPRIVATE);}
"protected"             {count(); return (TPROTECTED);}
"final"                 {count(); return (TFINAL);}
"abstract"              {count(); return (TABSTRACT);}
"static"                {count(); return (TSTATIC);}
"class"                 {count(); return (TCLASS);}
"interface"             {count(); return (TINTERFACE);}
"extends"               {count(); return (TEXTENDS);}
"implements"            {count(); return (TIMPLEMENTS);}
"super"                 {count(); return (TSUPER);}
"this"                  {count(); return (TTHIS);}
"throw"                 {count(); /*return (TTHROW);*/}
"throws"                {count(); /*return (TTHROWS);*/}
"try"                   {count(); /*return (TTRY);*/}
"catch"                 {count(); /*return (TCATCH);*/}
"if"                    {count(); return (TIF);}
"else"                  {count(); return (TELSE);}
"false"                 {count(); return (TFALSE);}
"true"                  {count(); return (TTRUE);}
"switch"                {count(); return (TSWITCH);}
"case"                  {count(); return (TCASE);}
"continue"              {count(); return (TCONTINUE);}
"break"                 {count(); return (TBREAK);}
"default"               {count(); return (TDEFAULT);}
"for"                   {count(); return (TFOR);}
"while"                 {count(); return (TWHILE);}
"do"                    {count(); return (TDO);}
"instanceof"            {count(); /*return (TINSTANCEOF);*/}
"finally"               {count(); /*return (TFINALLY);*/}
"new"                   {count(); return (TNEW);}
"null"                  {count(); return (TNULL);}
"return"                {count(); return (TRETURN);}
"synchronized"          {count(); /*return (TSYNCHRONIZED);*/}
"printString"           {count(); return (TPRINTS);}
"printInt"              {count(); return (TPRINTI);}
"printFloat"            {count(); return (TPRINTF);}
"print"                 {count(); return (TPRINT);}
"main"                  {count(); return (TMAIN);}
"package"               {count(); return (TPACKAGE);}

/*--Partie qui permet de reconnaitre les midentificateurs pour les renvoyer au parser --*/
/*--Cette partie vient après la partie des mots clés OBLIGATOIREMENT  ---*/
```

```
/*--afin de permettre la reconnaissance des mots clés autant que mots clés---*/
/*--et non pas des identificateurs----*/
{ID}                    {count(); return strToken(ID); }

 /*--La partie qui permet de reconnaitre toute sorte de caractères --*/
 /*--appartenant au lanagage kawa qui ne sont ni entier ni réel ni identificateur ni mot clé--*/
 /*--Exemple: les parenthèses, les opérateurs, etc... --*/
\"[^\n\"]*\"            {count(); return (STRING);}
"+"                     {count(); return ('+');}
"-"                      {count(); return ('-');}
"*"                     {count(); return ('*');}
"/"                     {count(); return ('/');}
"%"                     {count(); return ('%');}
"="                     {count(); return ('=');}
"+="                    {count(); return (TPLUSEQ);/**/}
"-="                    {count(); return (TMINUSEQ);/**/}
"*="                    {count(); return (TMULEQ);/**/}
"/="                    {count(); return (TDIVEQ);/**/}
"%="                    {count(); return (TMODEQ);/**/}
"++"                    {count(); return (TINC);}
"-"                    {count(); return (TDEC);}
"=="                    {count(); return (TCEQ);}
"!="                    {count(); return (TCNE);}
"<"                     {count(); return ('<');}
"<="                    {count(); return (TCLE);}
">"                     {count(); return ('>');}
">="                    {count(); return (TCGE);}
"||"                    {count(); return (TOR);}
"&&"                    {count(); return (TAND);}
"!"                     {count(); return ('!');}
"&"                     {count(); return ('&');}
"&="                    {count(); return (TANDBINEQ);/**/}
"|"                     {count(); return ('|');}
"|="                    {count(); return (TORBINEQ);/**/}
"^"                      {count(); return ('^');}
"^="                     {count(); return (TXORBINEQ);/**/}
"«"                     {count(); return (TDECAL);/**/}
"«="                    {count(); return (TDECALEQ);/**/}
"»"                     {count(); return (TDECAR);/**/}
"»="                    {count(); return (TDECAREQ);/**/}
"»>"                    {count(); return (TDECALNS);/**/}
"»>="                   {count(); return (TDECALNSEQ);/**/}
"("                      {count(); return ('(');}
")"                      {count(); return (')');}
"{"                      {count(); return ('{');}
```

```
"}"                     {count(); return ('}');}
"["                     {count(); return ('[');}
"]"                     {count(); return (']');}
"."                     {count(); return ('.');}
","                     {count(); return (',');}
";"                     {count(); return (';');}
"~"                     {count(); return ('~');}
"?"                     {count(); return ('?');}
":"                     {count(); return (':');}
"//".*\n                {count();}
"/*"[^*\/]*"*/"         {count();}
[\n]                    {count();}
[ \t\v\f]               {count();}
.                       {count(); printf("TOKEN inconnu!\n"); yyterminate();}
```

2 Analyse syntaxique

2.1 Objectif de l'analyse syntaxique

l'objectif de l'analyse syntaxique est de reconnaître les phrases appartenant à la syntaxe du langage son entrée est le flot des lexèmes construits par l'analyse lexicale, sa sortie est un arbre de syntaxe abstraite représentant le programme. Elle permet d'identifier et de localiser les erreurs de syntaxe du langage pour notre cas le langage source est le langage jouet kawa.

2.2 Grammaire du langage kawa

Afin de spécifier et de générer le langage kawa nous avons défini une grammaire non ambigüe permettant d'engendrer toute la syntaxe du langage source, cette grammaire permettra par la suite de construire l'arbre de syntaxe abstraite qui lui sera utile pour la prochaine phase d'analyse sémantique.
Notre grammaire est constituée d'un ensemble de règles de productions :

```
/*************** Grammaire ***************************/

Program  ->  Package ImportDeclaration Modifiers ClassDeclaration
| Package ImportDeclaration Modifiers InterfaceDeclaration

Package  ->  TPACKAGE ID ';'
/*----La partie qui gère les identificateurs et les identificateurs FQN   -----*/
Ids  ->  '.' ID Ids
| Epsilone

QList  ->  ',' ID Ids QList
| Epsilone

ListIds  ->  ID Ids QList

/*----les tableaux------*/
/*----Cette partie n'est gérée que syntaxiquement ------*/
Type  ->  BasicType Tables
| ID Ids Tables

Tables  ->  '[' ']' Tables
| Epsilone

TablesIndexe  ->  '[' ENTIER ']' TablesIndexe
| Epsilone

/*----les types primitifs (Nous avons pris string comme un type primitif)------*/
BasicType  ->  TBYTE
| TSHORT
| TCHAR
| TINT
```

```
| TLONG
| TFLOAT
| TDOUBLE
| TBOOLEAN
| TSTRING

/*------- Partie des modificateurs--*/
Modifier  ->  TPUBLIC
| TPROTECTED
| TPRIVATE
| TSTATIC
| TABSTRACT
| TFINAL

Modifiers  ->  Modifiers Modifier
| Epsilone


Static -> TSTATIC
| Epsilone

/*---- Sous-Partie qui complete la partie des imports ----*/
All  ->  '.' ID All
| '.' '*'
| Epsilone

/*---- Partie d'héritage des classes----*/
Extends  ->  TEXTENDS Type
| Epsilone

/*---- Partie d'héritage des interface----*/
ExtendsList  ->  TEXTENDS ListIds
| Epsilone

/*---- Partie d'implémentation des interface----*/
Implements  ->  TIMPLEMENTS ListIds
| Epsilone


/*-----IMPORTS -------*/
ImportDeclaration  ->  TIMPORT ID All ';' ImportDeclaration
| Epsilone

/*------ Entête des classes et des interfaces------*/
ClassDeclaration  ->  TCLASS ID Extends Implements ClassBody
```

```
InterfaceDeclaration  ->  TINTERFACE ID ExtendsList InterfaceBody

/*------Corps d'une classe---------*/
ClassBody ->  '{' MemberDecs '}'

MemberDecs -> MemberDecs MemberDec
| Epsilone


MemberDec: Modifiers Type ID VariableInitializer VariableDeclaratorList ';'
| Modifiers Type ID FormalParameters Block
| Modifiers TVOID ID FormalParameters Block
| Modifiers ID FormalParameters Block


/*---- Corps d'une interface -------*/
InterfaceBody ->  '{' Prototypes '}'

Prototypes -> Prototypes Prototype
| Epsilone

Prototype -> Modifiers Type ID FormalParameters';'
| Modifiers TVOID ID FormalParameters ';'

/*------------------- partie de parametres des methodes ----------------------*/
FormalParameters -> '(' FormalParameterDecls ')'
| VoidFormalParametrs


VoidFormalParametrs -> '(' ')'


FormalParametersCalledMethod -> '(' FormalParametersCalledMethodDecls ')'
| VoidFormalParametrs
;
FormalParametersCalledMethodDecls -> ID '[' ENTIER ']' TablesIndexe FormalParameterCalledMethodDeclsRes
| Expression FormalParameterCalledMethodDeclsRest


FormalParameterDecls -> VariableModifiers Type VariableDeclaratorId FormalParameterDeclsRest


VariableModifiers -> VariableModifiers VariableModifier
| Epsilone
```

```
VariableModifier: TFINAL
| TVALUE


FormalParameterDeclsRest -> ',' FormalParameterDecls
| Epsilone


FormalParameterCalledMethodDeclsRest -> ',' FormalParametersCalledMethodDecls
| Epsilone


VariableDeclaratorId -> ID Tables


/*----- Partie des blocks et les variables locales ------*/
Block ->  '{' BlockStatements '}'

BlockStatements ->  BlockStatements BlockStatement
| Epsilone

BlockStatement ->  Print
| PrintS
| PrintF
| PrintI
| ID '.' ID Ids Tables VariableDeclarator VariableDeclaratorList ';'
| ID Tables VariableDeclarator VariableDeclaratorList ';'
| BasicType Tables VariableDeclarator VariableDeclaratorList ';'
| ID TablesIndexe '=' Expression ';'
| Expression ';'
| TTHIS '.' VariableDeclarator ';'
| TTHIS '.' ID FormalParametersCalledMethod ';'
| TSUPER FormalParametersCalledMethod ';'
| Statement
| ';'

/*----- Partie des print -----*/
Print  ->  TPRINT '(' Args ')' ';'

Args  -> factFinal ArgsRest
| Epsilone

ArgsRest  ->  '+' factFinal ArgsRest
```

```
| Epsilone

PrintF  ->  TPRINTF '(' ArgsF ')' ';'

ArgsF  ->  REEL
| ID
| Epsilone

PrintI  ->  TPRINTI '(' ArgsI ')' ';'

ArgsI  ->  ENTIER
| ID
| Epsilone

PrintS  ->  TPRINTS '(' ArgsS ')' ';'

ArgsS  ->  STRING
| ID
| Epsilone

/*----- Partie des Statements ------*/
Statement ->  Block
| TIF '(' Expression ')' Statement
| TIF '(' Expression ')' Statement TELSE Statement
| TSWITCH '(' Expression ')' '{' SwitchBlockStatementGroups '}'
| TWHILE '(' Expression ')' Statement
| TDO Statement TWHILE '(' Expression ')' ';'
| TFOR '(' ForControl ')' Statement
| TBREAK ID ';'
| TBREAK ';'
| TCONTINUE ID ';'
| TCONTINUE ';'
| TRETURN Expression ';'
| TRETURN ';'

/*--------------------Partie switch-----------------------*/
SwitchBlockStatements -> SwitchBlockStatements SwitchBlockStatement
| Epsilone


SwitchBlockStatement: TCASE Expression ':' BlockStatements
|  TDEFAULT ':' BlockStatements


/*--------------- partie for ---------------------------------*/
```

```
ForControl -> ForVarControl ';' Expression ';'
| ForVarControl ';' Expression ';' ForUpdate
| ForVarControl ';' ';'
| ForVarControl ';' ';' ForUpdate
| ';' Expression ';'
| ';' Expression ';' ForUpdate
| ';' ';'
| ';' ';' ForUpdate


ForVarControl -> Type VariableDeclaratorId VariableDeclaratorList
| Type VariableDeclaratorId '=' Expression VariableDeclaratorList
| ID Ids VariableDeclaratorList
| ID Ids '=' Expression VariableDeclaratorList



ForUpdate -> Expression StatementExpressionList
|  ID Ids TablesIndexe '=' Expression  StatementExpressionList


StatementExpressionList -> ',' Expression StatementExpressionList
| ','  ID Ids TablesIndexe '=' Expression  StatementExpressionList
| Epsilone


/*-----------Partie Expression ------------*/
/*--Cette partie est gérée de cette manière afin de conserver l'ordre de priorité---*/
Expression  ->  FacteurAffect

/*----- Partie des opérateurs concaténés avec "=" ------*/
FacteurAffect  ->  FacteurEffect TORBINEQ ExpressionOr
| FacteurEffect TXORBINEQ ExpressionOr
| FacteurEffect TANDBINEQ ExpressionOr
| FacteurEffect TDECALNSEQ ExpressionOr
| FacteurEffect TDECAREQ ExpressionOr
| FacteurEffect TDECALEQ ExpressionOr
| FacteurEffect TMODEQ ExpressionOr
| FacteurEffect TDIVEQ ExpressionOr
| FacteurEffect TMULEQ ExpressionOr
| FacteurEffect TMINUSEQ ExpressionOr
| FacteurEffect TPLUSEQ ExpressionOr
| ExpressionOr

/*-----------Sous-partie des Expressions de comparaisons ------------*/
```

```
ExpressionOr   ->  ExpressionOr TOR ExpressionAnd
| ExpressionAnd

ExpressionAnd  ->  ExpressionAnd TAND ExpressionOrLogic
| ExpressionOrLogic

/*--Cette sous partie des Expressions de comparaison n'est gérée que syntaxiquement---*/
ExpressionOrLogic  ->  ExpressionOrLogic '|' ExpressionOrExLogic
| ExpressionOrExLogic

ExpressionOrExLogic  ->  ExpressionOrExLogic '\^' ExpressionAndLogic
| ExpressionAndLogic

ExpressionAndLogic  ->  ExpressionAndLogic '\&' ExpressionEqNeq
| ExpressionEqNeq

/*--Cette sous partie des Expressions de comparaison est gérée syntaxiquement et sémantiquement
---*/
ExpressionEqNeq   ->  ExpressionEqNeq TCNE ExpressionCompEq
| ExpressionEqNeq TCEQ ExpressionCompEq
| ExpressionCompEq

ExpressionCompEq  ->  ExpressionCompEq TCGE TermePlus
| ExpressionCompEq '>' TermePlus
| ExpressionCompEq TCLE TermePlus
| ExpressionCompEq '<' TermePlus
| TermeDecal

/*-----------Sous_partie des Expressions de décalage ------------*/
/*--Cette sous partie des Expressions de décalage n'est gérée que syntaxiquement---*/
TermeDecal  ->  TermeDecal TDECALNS TermePlus
| TermeDecal TDECAR TermePlus
| TermeDecal TDECAL TermePlus
| TermePlus

/*---Sous_partie des Expressions Arithmétiques --------*/
TermePlus     ->  TermePlus '+' terme
| TermePlus '-' terme
| terme

terme   ->  terme '*' facteur
| terme '/' facteur
| terme '%' facteur
| facteur
```

```
/*------ Sous_partie des opérateurs unaires ------------*/
facteur  ->  '~' facteur
| '!' facteur
| '-' facteur
| '+' facteur
| ID TDEC
| TDEC ID
| ID TINC
| TINC ID
| factFinal

/*-- Sous_partie des constantes et des références pour tout type d'opération et d'affectation--*/
factFinal ->  '(' Expression ')'
| ENTIER
| REEL
| ID Ids
| STRING
| ObjectInitializer
| MethodeInitializer
| TTRUE
| TFALSE
| TNULL

/*------ Sous_partie d'initialisation des objets-------*/
ConstructorCall -> TNEW ID Ids FormalParametersCalledMethod


MethodCall -> ID FormalParametersCalledMethod


ArrayInitializer -> TNEW BasicType '[' ENTIER ']'
| TNEW ID Ids '[' ENTIER ']'


LinkedMethodVarCall -> TTHIS '.' ID LinkedMethodVarCallList
| ID LinkedMethodVarCallList
| TTHIS '.' MethodCall LinkedMethodVarCallList
| MethodCall LinkedMethodVarCallList


LinkedMethodVarCallList -> '.' ID LinkedMethodVarCallList
| '.' MethodCall LinkedMethodVarCallList
| Epsilone
```

2.3 Construction de l'arbre abstrait

Nous construisons à travers cette phase d'analyse un arbre abstrait de syntaxe qui sera utilisé par les prochaines phases d'analyse (la sémantique et la génération de code intermédiaire). L'analyse sémantique a pour objectif de vérifier le sens de l'arbre monté en mémoire et de le décorer en ajoutant des informations nécessaires pour la génération de code intermédiaire.

Dans le cadre de ce projet nous avons opté pour une solution qui produit un seul arbre en mémoire **KawaTree** qui est une collection de classe. Ce même KawaTree sera utilisé par l'ensemble des modules, chacun de ces modules possèdent une interface de connexion avec l'arbre pour extraire les informations nécessaires à cette phase d'analyse.

Notre objectif pendant cette phase est de construire et de produire le KawaTree afin de permettre aux autres modules de continuer la chaine de compilation en prenant en paramètre d'entrée le KawaTree qui est un arbre abstrait commun pour tous les modules

Nous présentons le diagramme de classe (KawaTree)ainsi que des classes qui ont une relation avec ce diagramme mais qui ne sont utilisées que par le module syntaxique :

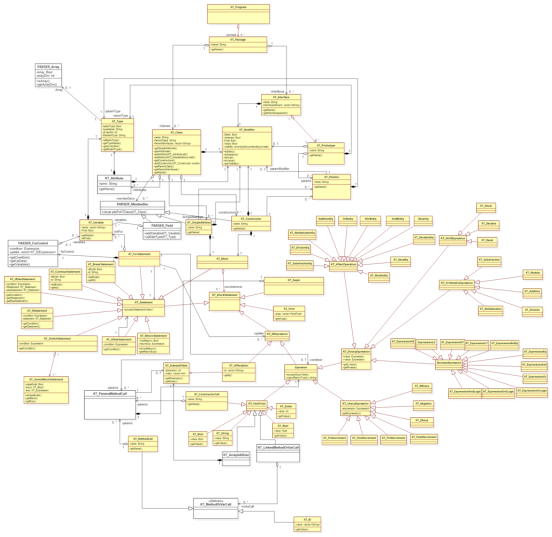

FIGURE 1 – KawaTree avec des classes du parser.

2.4 Outils de réalisation

Afin de réaliser le module d'analyse syntaxique nous avons utilisé les deux outils :

- **Flex** : c'est une version de lex qui est un générateur d'analyse lexical, nous pouvons définir à travers cet outil des unités lexicales pour les reconnaitre après dans un processus de compilation d'un programme source.
- **GNU Bison** : est l'implémentation GNU du compilateur de compilateur yacc, spécialisé dans la génération d'analyseurs syntaxiques, il permet de définir la grammaire du langage source ainsi que de produire un analyseur syntaxique en déclenchant les actions des règles de productions invoquées par le programme source, les règles de production sont déclenchés de bas vers le haut car bison est basé sur un analyse ascendante c'est la méthode d'analyse la plus performante.

Nous utiliserons ces deux outils Flex et Bison dans cette phase d'analyse syntaxique car le bison prend en paramètre d'entré les lexèmes qui ont été défini dans Flex afin de produire un arbre abstrait de syntaxe ce dernier sera généré en c++.