FLEX et BISON

tp.l

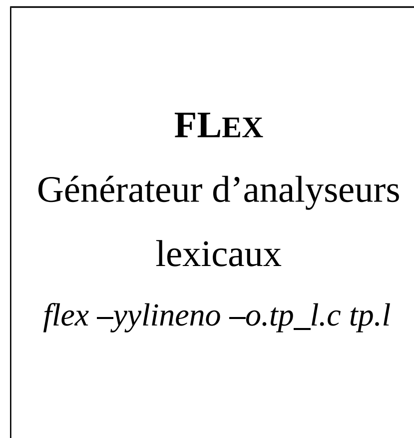
- abréviations
- expressions régulières et actions associées.

tp.h :

- définition de types
- Étiquettes pour l'AST

tp_y.h :
Codes des tokens

→



→

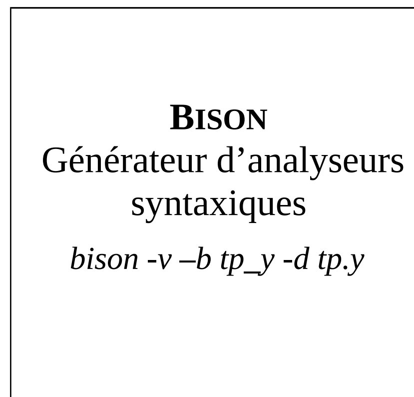
tp_l.c :

analyseur lexical =
la fonction `yylex()`

- reconnait le prochain token dans le source
- exécute l'action associée
- gère le numéro de ligne dans le fichier source : variable `yylineno`

tp.y :
- interface avec FLEX : `%token` …
- règles de précédence : `%left` …
- typage des attributs: `%type` …
- grammaire et règles de calcul des attributs associés aux productions

→



→

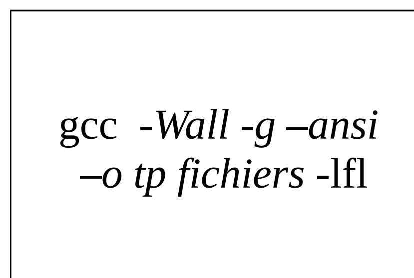
tp_y.c :

analyseur syntaxique =
la fonction `yyparse()`
- appelle `yylex()`
- exécute les actions associées aux réductions

tp_y.output :
automate LR(0) + conflits
tp_y.h : constantes (tokens)

- **tp_l.c (A. L.)**
- **tp_y.c (A. S.)**
- **tp_y.h**
- **tp.h, tp.c**, … :
fonctions codant vos actions sémantiques

→



→

tp : l'exécutable.

`main()` doit appeler `yyparse()` qui lance l'analyse syntaxique et exécute vos actions.

Pour le TP, les squelettes de certains fichiers sont fournis, ainsi qu'un `makefile`. Le fichier **`tp_y.h`** est produit par Bison et contient des constantes C pour les tokens. **Ne pas modifier ce fichier à la main car il est écrasé à chaque appel à Bison.**

Le fichier **tp.h** contient des définitions de types et des « étiquettes » utiles pour annoter les sommets des arbres de syntaxe abstraite. **Attention à ne pas écraser une des constantes correspondant à un token dans tp_y.h**

Un squelette de fichier TP.l pour FLEX

Cette partie définit des abréviations utilisables dans la suite du fichier. **À compléter pour le TP**

```
Chiffre     [0-9]
Entier      {Chiffre}+
Lettre      [a-zA-Z]
LC          ({Lettre}|{Chiffre})
Delim       [ \n\t\r]
```

Attention au symbole `'-'` qui indique un intervalle de valeurs et non pas le symbole lui-même.

```
%{          /* la partie entre %{ et %} sera recopiée dans le fichier tp_l.c */
#include "tp.h"
#include "tp_y.h"
#include <string.h>

extern YYSTYPE yylval;
extern int yylineno;

int ma fonctionPreferee(char *s) {  /* a completer */  }

/* Les codes symboliques pour les tokens (comme CSTE ou RELOP) sont définis dans tp_y.h
 * Les étiquettes pour l'arbre de syntaxe abstraite sont définies dans tp.h
 */
%}
```

La partie ci-dessous définit une partie dles tokens à reconnaître. **Elle est bien sûr à compléter ! Seul ce qui apparaît ci-dessous contribue à l'automate utilisée par l'analyseur lexicale.**

```
%%
{Entier}         { yylval.I = atoi(yytext); return(CSTE); }
{Lettre}{LC}*    { yylval.S = strdup(yytext) ; return ID; }
{Delim}+         { /* ignorer les delimiteurs */ }
```

En détail

Exportées

```
char *yytext:    texte du token courant (écrasé à chaque appel à yylex)
int yylen :      la longueur du texte ci-dessus.
int yylineno:    le compteur du nombre de lignes
int yylex():     la fonction renvoie le code symbolique du prochain token
```

Importées

yylval : *variable **de type YYSTYPE** pour stocker la « valeur » d'un token(interface de flex avec l'extérieur).*
Le type YYSTYPE est défini dans tp.h comme une « union » C.

Le paramètre d'édition de liens `–lfl` fournit des versions par défaut des fonctions `yywrap()` et `main`().

Les principales notations pour `FLEX`

Pour ***les expressions régulières*** *(tokens) :*

```
e*                    un nombre quelconque d’occurrences de e
e+                    un nombre positif d’occurrences de e
e?                    une occurrence optionnelle de e
rs                    une occurrence de r suivie d’une occurrence de s
r|s                   une occurrence de r ou une occurrence de s
{abreviation}         remplace l’abréviation par sa définition
.                     n’importe quel caractère sauf le retour chariot ‘\n’
```

Attention : les caractères ‘"’ et ‘<’ (en début d’expression) ont une signification particulière, de même que ‘.’ Les faire précéder par "\" si besoin.

Pour les ***abréviations*** *:*

`[abc]` le caractère `‘a’` ou `‘b’` ou `‘c’`
`[a-zA-Z]` les symboles compris entre `‘a’` et `‘z’` ou entre `‘A’` et `‘Z’`
`[^abc]` n’importe quel caractère sauf `‘a’`, `‘b’` et `‘c’`

Le caractère `‘\’` permet d’enlever sa signification particulière au caractère qui le suit
`AntiSlash` `\\`

En cas d’ambiguïté de reconnaissance, `yylex()` privilégie la reconnaissance par l’unité qui reconnaît le préfixe de texte **le plus long**.
À tailles égales, l’unité qui **apparaît en premier dans le fichier** `tp.l` est privilégiée.

La fonction `yylex()` renvoie `0` quand elle a reconnu l’unité correspondant **à la fin de fichier**.

Vous devez reconnaître tous les caractères, ne serait-ce que pour signaler qu’ils sont illégaux (par exemple un `‘!’` dans notre tp). Par défaut, yylex() **recopie sur sa sortie standard tout caractère non reconnu** par une des expressions régulières. Des **affichages parasites** sont le signe de la non reconnaissance par l'analyseur lexical des caractères ainsi affichés.

Principes de construction d'un interprète sous **Bison**

Bison est un analyseur syntaxique **ascendant**, implémentant l'analyse LaLR(1) qui étend la méthode SLR(1) vue en cours. En plus des aspects syntaxiques, Bison permet de définir les règles de calcul d'un **unique attribut synthétisé**. Une production de la forme

$X:\ X_1\ X_2\ \ldots\ X_n$ { /* fragment de code C */ }

définit une règle de grammaire à laquelle on a attaché une action, c'est-à-dire un bloc d'instructions en langage C qui sera exécuté au moment de procéder à une **réduction** par cette production. Cette action dépend habituellement des « valeurs » de l'attribut pour les symboles de la partie droite ; ces valeurs auront déjà été calculés puisque l'analyse est ascendante.Bison ne fournit de mécanisme d'évaluation que pour un unique attribut synthétisé mais qui peut être arbitrairement complexe. Si on a besoin d'un d'attribut **hérité**, on doit le simuler par une **variable globale** gérée dans les actions des productions.

Attribut : En parallèle avec la **pile d'analyse** syntaxique, Bison maintient une « **pile de valeur** », qui correspondent aux valeurs de cet attribut. Par convention, **`$$`** désigne l'attribut de la partie gauche de la production et **`$i`** l'attribut du **i**[ème] symbole de la partie droite. En général, une « action » contient donc une instruction de la forme **`$$ = f($1, …, $n)`**. Bison dépilera les valeurs associées aux valeurs **`$1`, …, `$n`** de la partie droite de règle et empilera la valeur associée à **`$$`**. Un tel mécanisme peut servir à construire un arbre de syntaxe abstraite : **`f`** créera un arbre en ajoutant une racine d'étiquette donnée (dépendante de la règle) aux sous-arbres associés aux symboles de la partie droite. Une production sans action correspond à une action par défaut **`$$ = $1;`** qui peut entraîner des erreurs à la compilation.

Un non-terminal n'a pas toujours d'action ou de valeur associée : soit il n'a pas d'action, soit il procède par « effet de bord » plutôt qu'en calculant une valeur stockée dans la pile de valeurs**. Cependant toutes les productions d'un même non-terminal doivent être cohérentes de ce point de vue.**

Typage des attributs : les attributs des divers non-terminaux n'ont pas tous le même type de valeurs. Cependant, le code C engendré par Bison doit définir un type du langage C pour pouvoir déclarer la variable contenant la pile des valeurs ! Ce type, nommé YYSTYPE, est défini comme une « union » et vous devez ajoutez à la définition de ce type les « variantes » nécessaires pour les types de vos attributs. Une variable d'un type union devant toujours être suffixée par le nom de la variante de l'union utilisée à cet endroit du programme, Bison prévoit une section %type pour préciser quel non-terminal utilise quelle variante de l'union et s'en sert ensuite pour ajouter automatiquement le suffixe. Les clause ci-dessous indiquent que les non-terminaux X et Y sont associés à un attribut dont le type est celui de la variante S dans YYSTYPE, tandis que Z utilise la variante T du type :

```
%type <S> X Y
%type <T> Z
```

Application au TP : Principe de construction d’un interprète

Dans le langage considéré, un programme consiste en une liste (éventuellement vide) de déclarations de variables avec leur initialisation suivie une expression finale. Les détails sont donnés dans l’énoncé.

```
x := 3;
y := 12 + x;
t := if z > y then x + 3 else y + 3;
begin
    2 * if t = z then x * y + t else 1 * y
end
```

1ère méthode : construction et parcours d’arbres de syntaxe abstraite

La manière la plus simple construit lors de l’analyse syntaxique **un arbre de syntaxe abstraite (AST)** qui représente les constructions du programme source (déclaration, liste de déclarations, expressions, etc.). Une fois l’arbre construit, on peut le parcourir à volonté pour procéder aux vérifications contextuelles **puis** à l’évaluation des expressions. Pour cette partie, qui se résume à un parcours récursif d’arbre basé sur l'étiquette associée à chaque nœud de l’AST, on doit fournir l’« environnement » courant dans lequel trouver la valeur de chaque identificateur, c’est-à-dire l’ensemble des couples (variable, valeur) pour les variables visibles en cet endroit du programme. Cet environnement est aussi nécessaire pour les vérifications contextuelles puisqu’on doit vérifier qu’une expression ne référence que des variables déjà déclarées (et une variable ne peut être définie qu’une seule fois).

Le fichier `tp.h` contient des définitions de types C pour définir de tels sous-arbres et le fichier `tp.c` met à disposition des fonctions de construction : `makeTree`, `makeLeafString` et `makeLeafInt`.

2ème méthode : arbres de syntaxe abstraite avec évaluation au fil de l’eau

Une autre manière de faire ne construit pas l’arbre de tout le programme source : les règles du langage permettent d’évaluer l’expression d’une déclaration dès la fin de celle-ci car on ne peut référencer que des variables déjà déclarées, dont on connaît les valeurs. Une fois l’évaluation faite, on n’a plus besoin de l’arbre de l’expression : on ajoute le nouveau couple (variable, valeur) à l’environnement pour la suite de l’analyse du programme. On ne stocke donc que l’arbre de l’expression courante.

Pourrait-on se passer de toute construction d’arbre, en évaluant les expressions au fur et à mesure qu’on les reconnaît ? La réponse est « non », à cause du `if` qui ne doit évaluer que l’une des expressions des parties `then` et `else`. Il serait incorrect d’évaluer les deux avant que la règle du `if` choisisse laquelle renvoyer, comme on peut s’en convaincre avec l'expression `if 1=0 then 1/0 else 1` dont l’évaluation ne doit **pas** provoquer de division par zéro. Pour toute la partie « expression » on devra donc construire un AST qui représente cette expression et on ne procédera à son évaluation qu’une fois qu’on est sûr d’avoir une expression complète, c’est-à-dire au niveau de la règle associée à une déclaration, ou pour l’expression finale. Dans cette façon de faire, selon les règles de grammaire, les actions vont différer :

1. Pour la sous-grammaire des expressions, les actions vont construire un arbre qui symbolise l’expression reconnue, comme dans la première méthode.
2. Au niveau de chaque déclaration, on récupère l'arbre représentant l'expression de la partie droite, on effectue les vérifications contextuelles, on évalue l’expression représentée par l’arbre et on construit le couple (variable, valeur) qu’on ajoute à la liste courante.
3. Pour l’expression finale : on procède comme pour toute expression, on l’évalue en fonction des valeurs des variables puis on imprime le résultat global du programme source.

Remarque : un programme tel que `begin if 1=2 then x else 0` est **incorrect** puisqu’il contient une occurrence de la variable `x`, non déclarée, même si la valeur de cette variable n’intervient pas dans l’évaluation. **Les vérifications doivent être faites statiquement, avant l'exécution.**

Un squelette de fichier `TP.y` pour `Bison`

```
%token  If Then Else Begin End Affect ...
%token <S> ID
%token <I> CST
```

Interface avec Flex
définition d'un token et typage associé
(voir ci-dessous la clause `%type`)

```
 /* indications de précédence (en ordre croissant) et d'associativité.
  * Les opérateurs sur une même ligne ont la même précédence.
  * Les associativités disponibles sont %left, %right et %nonassoc
  * On peut faire apparaître un token ou un (et un seul) caractère littéral.
  * On peut définir des token « virtuels » pour leur associer une précédence
  * de manière à associer cette précédence à une production
  * (ajout en fin de production de %prec ce-token-virtuel).
  */
%left '+'
```

Indications de typage des « valeurs » associées aux actions sémantiques

```
%type <T> expr expr2
%type <V> expr3             /* voir YYSTYPE dans tp.h */
```

```
%{    /* partie recopiée telle quelle dans le fichier tp_y.c produit par Bison. */
#include <stdio.h>
#include "tp.h"
…
extern int yylex();            /*  la fonction d'anayse lexicale  */
%}
```

```
%%          /* L'axiome est la partie gauche de la première production */
S : declL BEG expr END  { action exécutée lors de la réduction }
;
```

Réfléchir si la récursion droite ou gauche convient mieux à vos actions

```
declL:                { action associée à une liste vide de déclarations  }
|       declL   decl  { action pour ajouter une déclaration à une liste }
;

decl : …         { traitement d'une déclaration }
;
```

*construction d'un **arbre de syntaxe abstraite**.*

```
expr : expr '+' expr  { $$ = makeTree(Eadd, 2, $1, $3); }
     | IF bexpr THEN expr ELSE expr { $$ = makeTree(ITE, 3, $2, $4, $6); }
     | ID { $$ = makeLeafStr($1); }
     | CST { $$ = makeLeafInt($1); }
```

Le type ***YYSTYPE*** (dans tp.h)

```
typedef union
{ char C;   /* pour flex */
  char *S;  /* les autres correspondent aux variantes utilisées */
  TreeP T;  /* dans les actions associées aux productions de    */
  int I;    /* la grammaire.                                    */
   …        /* Ajoutez d'autres variantes si besoin         */
} YYSTYPE;
```

En détail

Fichiers

```
tp_y.output :  version textuelle de l'automate LaLR(1).
tp_y.c :       le code de l'analyseur syntaxique.
tp_y.h :       Le fichier avec les constantes symboliques pour les tokens.
tp.h :         toutes les autres définitions nécessaires pour votre programme
```

Exportées

```
int yyparse()  lance l'analyse syntaxique et renvoie 0 en cas de succès et une
               valeur différente de 0 sinon.
YYSTYPE yylval : variable pour faire interface avecflex (transmission de la
               « valeur » d'un token), écrasée à chaque appel à flex.
```

Importées

YYSTYPE : le type des valeurs associées aux actions sémantiques.
int yylex(): l'analyseur lexical, appelé par yyparse() à chaque fois qu'il y a besoin d'un nouveau token
*void yyerror(char *s)*: appelée en cas d'erreur syntaxique

Le fichier tp_y.output

Il contient une description textuelle de l'automate LR(0) ainsi que des éventuels conflits décalage/réduction ('shit/reduce') et réduction/réduction(reduce/reduce).

```
Etat 28 conflits: 4 decalage/reduction
Etat 36 conflits: 4 decalage/reduction


Grammaire
    0 $accept: programme $end
    1 programme: declLO BEG expr END
    2 declLO: /* vide */
    3       | declL
...

Terminaux, suivis des règles où ils apparaissent

...

Non-terminaux, suivis des règles où ils apparaissent

...

etat 0
    0 $accept: . programme $end

    ID  decalage et aller a l'etat 1
    $defaut  reduction par utilisation de la regle 2 (declLO)

    programme  aller a l'etat 2
    declLO     aller a l'etat 3
    declL      aller a l'etat 4
    ...
```

```
etat 1

    7 var_decl: ID . AFF expr ';'

    AFF  decalage et aller a l'etat 7

etat 4

    3 declLO: declL .
    5 declL: declL . decl

    ID  decalage et aller a l'etat 1

    $defaut  reduction par utilisation de la regle 3 (declLO)

...

etat 28

    9 expr: expr ADD expr .
    9     | expr . ADD expr
   10     | expr . SUB expr
   11     | expr . MUL expr
   12     | expr . DIV expr

    ADD  decalage et aller a l'etat 19
    SUB  decalage et aller a l'etat 20
    DIV  decalage et aller a l'etat 21
    MUL  decalage et aller a l'etat 22

    ADD       [reduction par utilisation de la regle 9 (expr)]
    SUB       [reduction par utilisation de la regle 9 (expr)]
    DIV       [reduction par utilisation de la regle 9 (expr)]
    MUL       [reduction par utilisation de la regle 9 (expr)]
    $defaut    reduction par utilisation de la regle 9 (expr)

...

etat 36

    8 expr: IF bexpr THEN expr ELSE expr .
    9     | expr . ADD expr
   10     | expr . SUB expr
   11     | expr . MUL expr
   12     | expr . DIV expr

    ADD  decalage et aller a l'etat 19
    SUB  decalage et aller a l'etat 20
    DIV  decalage et aller a l'etat 21
    MUL  decalage et aller a l'etat 22

    ADD       [reduction par utilisation de la regle 8 (expr)]
    SUB       [reduction par utilisation de la regle 8 (expr)]
    DIV       [reduction par utilisation de la regle 8 (expr)]
    MUL       [reduction par utilisation de la regle 8 (expr)]
    $defaut   reduction par utilisation de la regle 8 (expr)
```