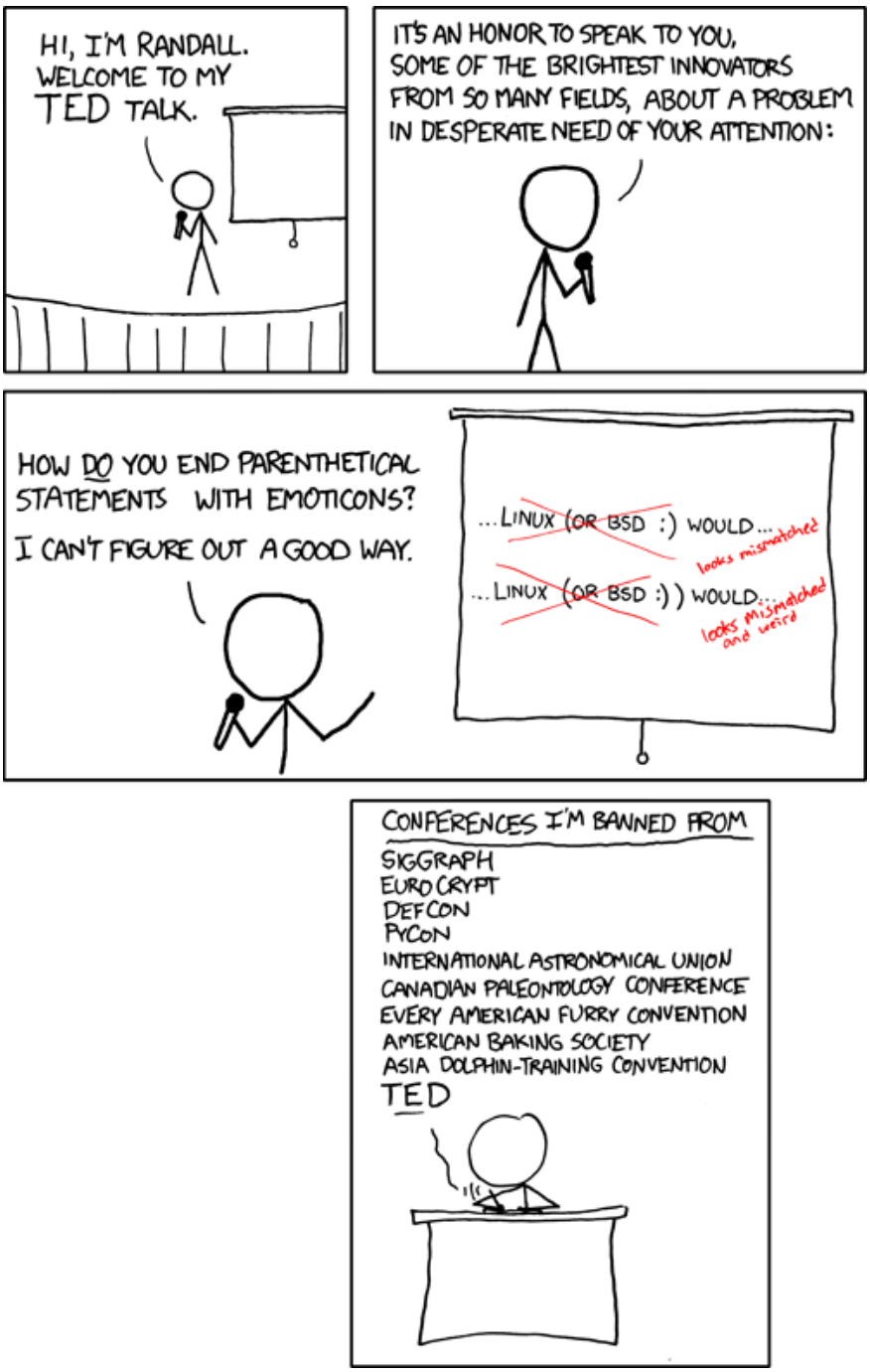
Van De Walle Bernard (A)
Francois Thibault (B)
Van Der Essen Frédéric (C)

INGI2132 : Langages et traducteurs

Rapport Final

Prof. B. Le Charlier



It is now possible with the HAPPY-) programming language!

Chapitre 1

Présentation de Happy :-)

1.1 Introduction

La langage happy, appelé ainsi parce qu'il est très permissif au niveau des caractères permit dans les identifiants comporte de nombreuses autres particularités. Tout d'abord, le langage ressemble très fort au LISP où tout est atom ou liste. Mais contrairement au LISP, notre langage est impératif et orienté objet. Un programme est une liste de méthode, une méthode est une liste dont le premier élément est le mot réservé fun le second une list d'argument et le dernier une liste de commande. Même principe pour le while et le if. Dans ce langage tout est fonction, dans le sens que toute instruction renvoie une valeur, y compris les if et le while qui renvoient 0, une fonction qui n'a pas d'instruction return renvoie null, le write renvoie la valeur qu'elle vient d'imprimer etc...

Les conditions aurait aussi renvoyé une valeur si le langage SLIP dans lequel notre langage est traduit le permettait.

1.2 Grammaire de départ

Cette grammaire est la grammaire exhaustive du langage si on rajoute que les id peuvent être formé de tout les caractères UTF-16 sauf des caractères réservés et qu'ils ne doivent pas être égale à un mot réservé.

```
<Program> ::= ( <Prog_list> )
<Prog_list> ::=  <Meth_or_fun> | <Prog_list> <Meth_or_fun>
<Meth_or_fun> ::=  <Method> | <Function>

<Function> := ( fun ( <Arglist> ) <Instr_list> )
<Arglist> ::=  id | id <Arglist>
<Method> ::= ( <Method_int> ( <Arglist> ) <Instr_list> )
```

```
<Instr_list> ::=  <Instr> | ( <Instr_list_np> )
<Instr_list_np> ::=  <Instr> | <Instr> <Instr_list_np>
<Instr> ::=  <Conditional> | <While_block> | <Call> | ( return <Expr> )

<Conditional> ::= ( if <Cond> <instr_list> <instr_list>  )
<Conditional> ::= ( if <Cond> <instr_list> )
<While_block> ::= ( while  <Cond> <Instr_list>  )
<call> ::=  <User_call> | <Method_call> | <Builtin_call>
<Builtin_call> ::=  <Assignment> | <Read_call> | <Write_call> | <Arithmetic_call>
<Assignment> ::= ( set id <Expr> )
<Assignment> ::= ( set <Id_int> <Expr> ) | ( set <This_int> <Expr> )
<Read_call>  ::= ( read )
<Write_call> ::= ( write <Expr> )
<Arithmetic_call> ::=  <Binary> | <Unary>
<Binary> ::= ( <Bin_id> <Expr> <Expr>  )
<Unary>  ::= ( <Un_id> <Expr>  )
<User_call>  ::= ( id  <Expr_list_np> ) | ( id )
<Expr_list_np> ::=  <Expr> | <Expr> <Expr_list_np>
<Method_call>  ::= ( <Id_id> <Expr_list_np>  ) |  ( <Id_id> )
<Method_call>  ::= ( <Super_id>  <Expr_list_np>  ) | ( <Super_id>  )
<Method_call>  ::= ( <This_id>  <Expr_list_np>  ) | ( <This_id> )
<Expr> ::= number | null | true | false | this | id
<Expr> ::= <Id_int> | <This_int> | <Instr>
<Cond> ::= <Rel> | ( <Log_bin_op> <Cond> <Cond> )
<Cond> ::= ( <Log_un_op> <Cond> <Cond> )
<Rel>  ::= ( <Rel_op> <Expr> <Expr> )
<Rel_op> ::= <= | >= | > |  < | =
<Log_bin_op> ::= and | or
<Log_un_op> ::= !
<Bin_id> ::= + | - | * | / | %
<Un_id> ::= neg
<Id_int> ::= id . number
<This_int> ::= this . number
<Super_id> ::= super . id
<This_id> := this . id
<Id_id> ::= id . id
<Method_int> ::= method . number
```

Cette grammaire n'est pas wp, mais elle exprime très bien ce qui est syntaxiquement correcte dans ce que nous avons réellement implémenté.

Voici quelque bout de code permis par cette grammaire :

```
(while (<= (3 * i) (9)) (write (set i (+ i 1))))
```

```
 (if (! (= i 9)) (return true) (return false))

(fun (++ i) (return (+ i 1)))

 (set i (++ i))
```

Chapitre 2

Mode d'emploi du compilateur

La version exécutable du compilateur et interpréteur Happy se trouve dans le jar exécutable *happy.jar*. Il prend 2 arguments obligatoire : la grammaire au format BNF et le programme en langage Happy. On peut rajouter *check* à la fin pour vérifier la grammaire en plus. Pour faire fonctionner les programmes Happy, il faut passer la grammaire *temp.bnf*. Pour vérifier n'importe quelle grammaire WP, il faut indiquer la grammaire un fichier programme bidon et enfin check. exemple :

```
java -jar happy.jar temp.bnf programme1.happy
java -jar happy.jar temp.bnf programme1.happy check
```

L'interpréteur sort les informations suivantes dans cet ordre, si le check de la grammaire est demandé, le résultat des tests et la table de précédence. Ensuite l'arbre sortit par le parser syntaxique après sa restructuration, ensuite la traduction de l'arbre en SLIP et puis (mais il semble que les retour à la ligne ne soit pas conforme ce qui donne des résultats étranges à la sortie dans le shell) la représentation en code interne SLIP. Et finalement la sortie du programme interprèté.

2.1 Les erreurs

Lorsqu'une erreur arrive dans n'importe quelle partie du compilateur l'erreur est affichée (avec plus ou moins de précision) et il s'arrête ensuite.

2.2 Les erreurs du vérificateur

2.3 Les erreurs du parser lexical

Il ne vérifie que deux choses. A la fin de l'analyse si il n'y a pas le même nombre de paranthèse ouvrantes que fermantes, l'erreur *Unexpected end.* Il est aussi capable de vérifier si à tout moment il y a trop de paranthèse fermantes, le messages est dès lors très explicite : *Too much)* .

2.4 Les erreurs de l'analyseur syntaxique

L'analyseur syntaxique signale trois erreurs. Lorsque deux termes ne peuvent se retrouver côte à côte, lorsqu'il n'arrive pas à trouver une règle pour réduire.

Et lorsqu'il a lu tout les caractères, si il reste plus d'un élément dans la pile une erreur est aussi renvoyé.

2.5 Les erreurs du traducteur

Les erreurs que renvoient le traducteur sont d'ordre sémantique. Le programme est valide syntaxiquement mais comme notre grammaire permet beaucoup de chose, Il faut remettre les choses en places avec le traducteur. Les erreurs sont du type : quelquechose est attendu dans ce type d'expr et là c'est pas le cas. exemple : textitExpected id after a set

2.6 Les erreurs de l'interpréteur

L'interpréteur reporte les erreurs d'exécutions telles que la division par 0, l'appel à méthode inconnue etc... Les erreurs sont clairement identifié dans le code interne structurée et reporté jusqu'au dessus de la pile d'exécution. Cela permet de retracer l'erreur sans trop de difficulté. Voici un exemple :

```
Error divide by 0
 in Cexpr a#1 / b#2
 at Ass A_0#3 := a#1 / b#2
 at CmdStmt [ lab9 : A_0#3 := a#1 / b#2 ; go to lab8]
 at divide/-1
 at Call divide
 at CmdStmt [ lab2 : A_0#3 := divide(#4, #5) ; go to lab1]
 at main/-1
```