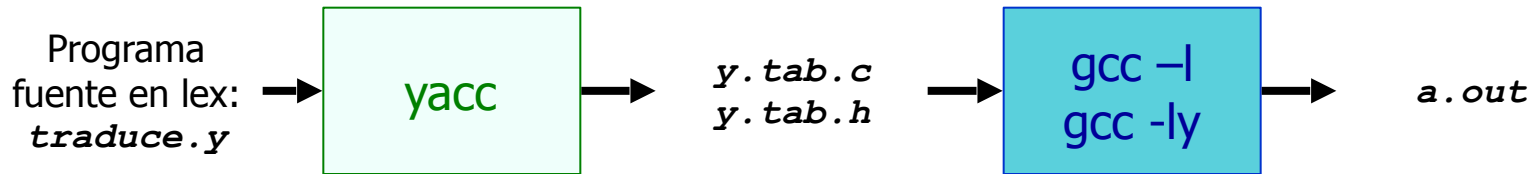




Yet Another Compiler Compiler
LALR(1) parser generator.



Funcionamiento de yacc



```
yacc -d traduce.y && gcc -o compiler y.tab.c -ly
```



```
y.tab.h contiene las definiciones de los tokens compartidas con LEX
```



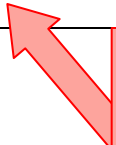
Estructura

```
%{  
Declaraciones  
%}  
Definiciones  
%%  
{producciones y reglas de traducción}  
%%  
{rutinas de C}
```



Definiciones

```
%token DIGIT PROGRAM STATEMENT  
%start PROGRAM
```



%token se definen los componentes léxicos compartidos entre LEX y YACC y que son generados en el archivo *y.tab.h* para importar desde LEX. Con *%start* se marca cual es token raíz de todo el árbol sintáctico.



Reglas de traducción (~BNF)

```
left_side :    right side
              {
                syntactic action
              }
              |    right side 2 { action C-snippet 2 }
              |    right side 3 { action C-snippet 3 }
              ;
```

Reglas de traducción

```
expr      :      termino '*' factor
          {
            $$ = $1 * $3;
          }
          |      factor
          ;
```

expr es un no-terminal (i.e. no está definido en las Definiciones como un token), que no hace falta declarar pero sí ubicar en el lado izquierdo de una producción eventualmente. **termino** y **factor** son terminales, tokens en el jargon de yacc.

Al aplicar la producción indicada se ejecuta la sentencia entre {} asociada a esa producción. \$1 indica el primer operador del lado derecho (término), \$2 el segundo ('*') y \$3 el tercero. Estos operadores son tomados del valor de la variable global yyval que se setea desde LEX cuando se reconoce ese lexema. \$\$ indica la referencia a la variable de valor asociada a expr.

Reglas de traducción

```
expr      :      termino '*' factor
{
  $$ = $1 * $3;
}
|      factor
;
```

\$\$ referencia al valor de yylval asociado al no terminal `expr` del lado izquierdo de la producción. `$n`, del lado derecho, representan los valores asociados a yylval para cada uno de los terminales y no terminales de ese mismo lado. Estos valores son asignados típicamente desde las reglas de las expresiones regulares establecidas en lex (los return de LEX). El tipo de dato de cada uno puede alterarse mediante el uso del definición `%union` ó `%type`.



Variables globales

<i>char *yytext</i>	Puntero al inicio del texto que se ha hecho coincidir con el patrón en LEX. Terminado con \0.
<i>yy leng</i>	Longitud de yytext (hasta el \0).
<i>yy lval</i>	Yacc lee esta variable para asignar los valores a cada uno de los terminales que obtiene de LEX. Es decir el valor de \$n coincide con lo asignado en yy lval justo antes de
<i>yy in yy out</i>	Punteros a la entrada y salida de los archivos del compilador.



Walkthrough sin LEX

```
%{
#include <ctype.h>
int yydebug=1;  // útil para debugging..
%}

%token DIGITO
%start linea
%%
linea      :      expr '\n'          { printf("%d\n", $1); }
           ;
expr       :      expr '+' termino { $$ = $1 + $3; }
           |      termino
termino    :      termino '*' factor { $$ = $1 * $3; }
           |      factor
           ;
factor     :      '(' expr ')'      { $$ = $2; }
           |      DIGITO
           ;
%% .....
```



Walkthrough

```
.....  
%%  
yylex() {  
    int c;  
    c = getchar();  
    if (isdigit(c) ) {  
        yylval = c-'0';  
        return DIGIT;  
    }  
    return c;  
}
```

Walkthrough

Esta primera sección de header se copia verbatim tal cual está en el programa .C que genera Yacc.

```
%{
#include <ctype.h>
int yydebug=1; // útil para debugging..
%}

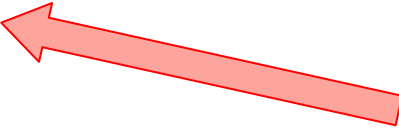
%token DIGITO
%start linea
%%
linea      :      expr '\n'          { printf("%d\n", $1); }
           ;
expr       :      expr '+' termino { $$ = $1 + $3; }
           |      termino
termino    :      termino '*' factor { $$ = $1 * $3; }
           |      factor
           ;
factor     :      '(' expr ')'      { $$ = $2; }
           |      DIGITO
           ;
%%
.....
```

Walkthrough

```
%{  
#include <ctype.h>  
int yydebug=1; // útil para debugging..  
%}
```

```
%token DIGITO  
%start linea  
%%
```

```
linea :      expr '\n'      { p  
      ;  
expr   :      expr '+' termino {  
      |      termino  
termino :      termino '*' factor  
      |      factor  
      ;  
factor :      '(' expr ')'  { $  
      |      DIGITO  
      ;  
%% .....
```

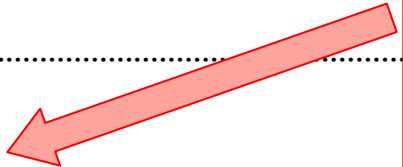


En este caso se define un único **no-terminal** DIGITO y se indica que de los no terminales el inicial es "**linea**". Cuando se ejecuta yacc, las definiciones de los terminales establecidas en esta sección van a parar como #define en el archivo y.tab.h que luego se tiene que importar desde LEX.



Walkthrough

```
.....  
%%  
yylex() {  
    int c;  
    c = getchar();  
    if (isdigit(c) ) {  
        yylval = c-'0';  
        return DIGIT;  
    }  
    return c;  
}
```



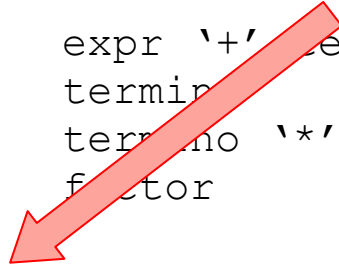
Este ejemplo no se compila junto con LEX sino que se provee una función `yylex` que está hecha "a mano". Esta función es la que ejecuta el compilador de yacc cada vez que necesita leer un nuevo TOKEN. Cuando se usa LEX, esta función es justamente la salida en código C que genera lex y que termina implementado un autómata finito más complicado y que puede reconocer todos los lexemas que se necesiten.

Walkthrough

```
%{
#include <ctype.h>
int yydebug=1; // útil para debugging..
%}

%token DIGITO
%start linea
%%
linea :      expr '\n' '\n", $1); }
      ;
expr :      expr '+' termino { $$ = $1 + $3; }
      |      termino
termino :   termino '*' factor { $$ = $1 * $3; }
      |      factor
      ;
factor :    '(' expr ')' { $$ = $2; }
      |      DIGITO
      ;
%%
.....
```

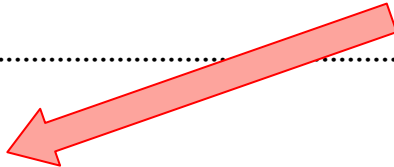
Así entonces cuando el parser de yacc necesita leer una nueva entrada, ejecuta 'yylex()'.





Walkthrough

```
.....  
%%  
yylex() {  
    int c;  
    c = getchar();  
    if (isdigit(c) ) {  
        yylval = c-'0';  
        return DIGIT;  
    }  
    return c;  
}
```



Se ejecuta entonces esta función, donde se identifica el nuevo token tomando su valor numérico y almacenando en la variable `yylval`. La función `yylex` le retornará al parser de Yacc cuál es el terminal identificado, en este caso DIGITO.

Walkthrough

```
%{  
#include <ctype.h>  
int yydebug=1; // útil para debugging..  
%}
```

```
%token DIGITO  
%start linea  
%%
```

```
linea :      expr '\n'  
      ;
```

```
expr :      expr '+' expr  
      |      termino  
termino :   termino '**'  
          |      factor  
          ;
```

```
factor :    '(' expr ')' { $$ = $2; }  
        |      DIGITO.   { $$ = $1; }  
        ;
```

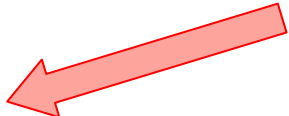
```
%% .....
```

Al retornar de yylex y recibir que el lexema identificado es DIGITO se REDUCE la producción "factor -> DIGITO" y se ejecuta el código del snippet asociado tomando el valor de \$1 directamente de lo que tiene almacenado en ese momento la variable yylval. En este caso, ese valor se sintetiza hacia el terminal factor (\$\$).



Tips

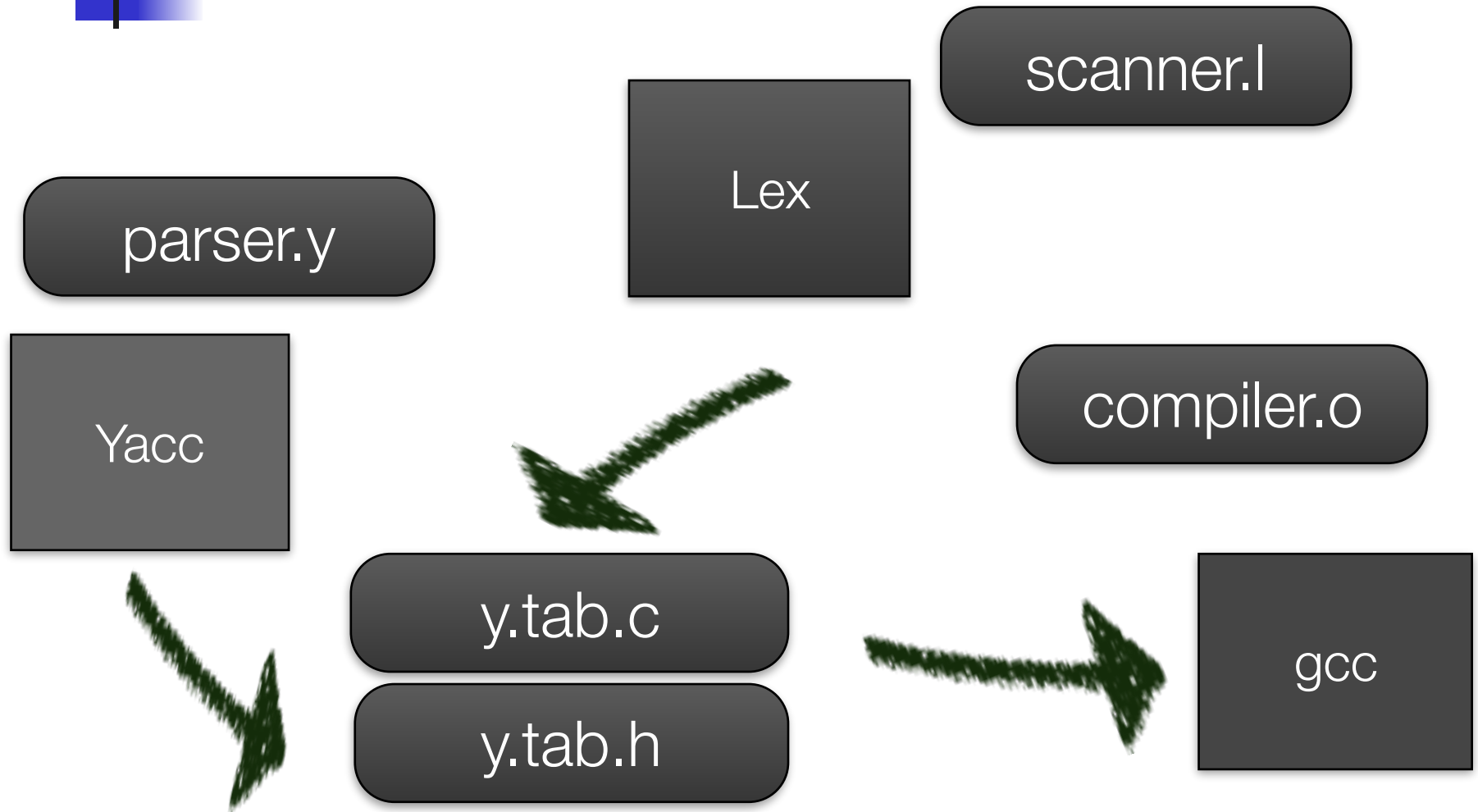
```
program : /* empty */  
        | statement  
        ;
```



Transiciones lamda

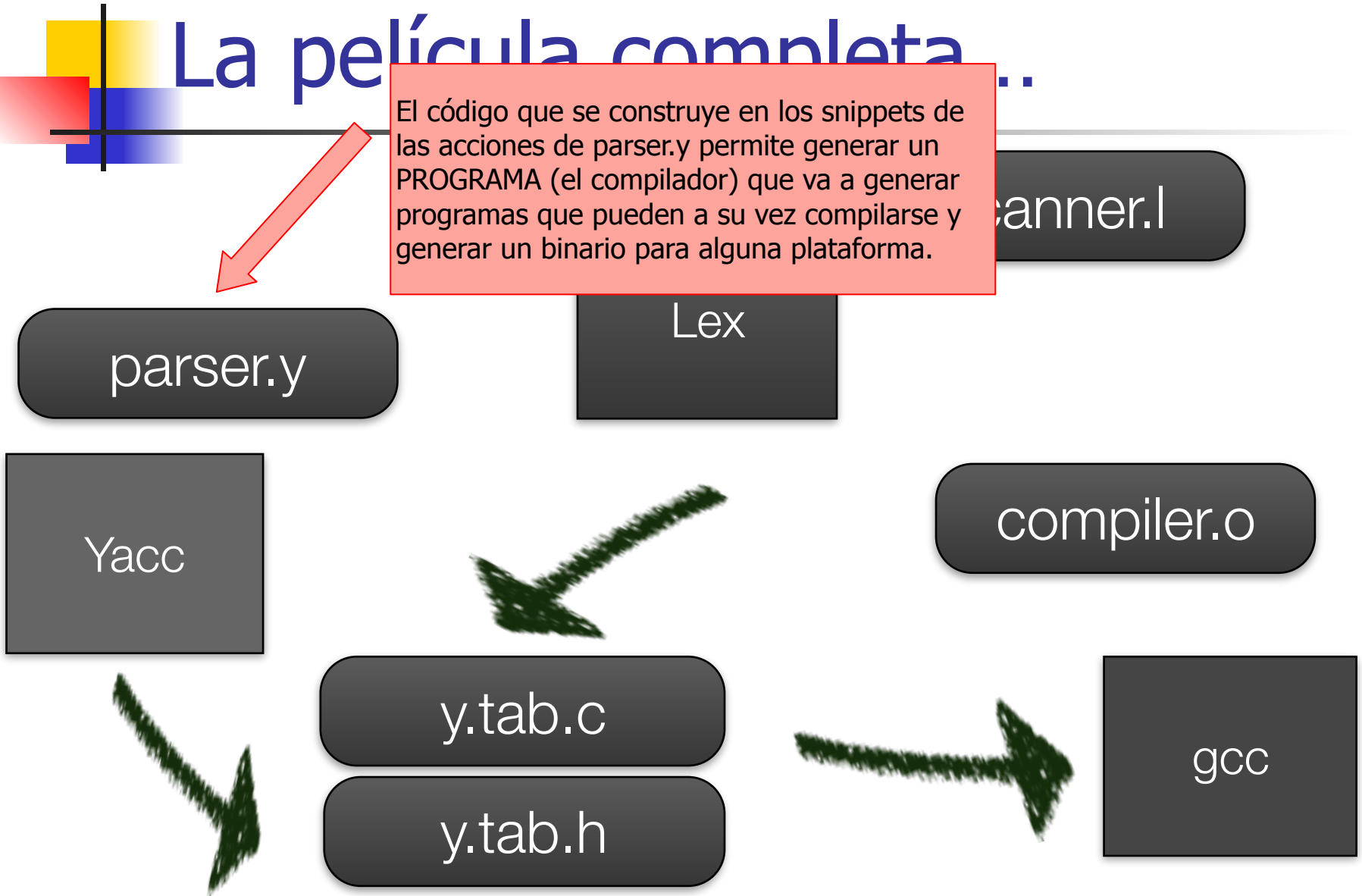


La película completa...

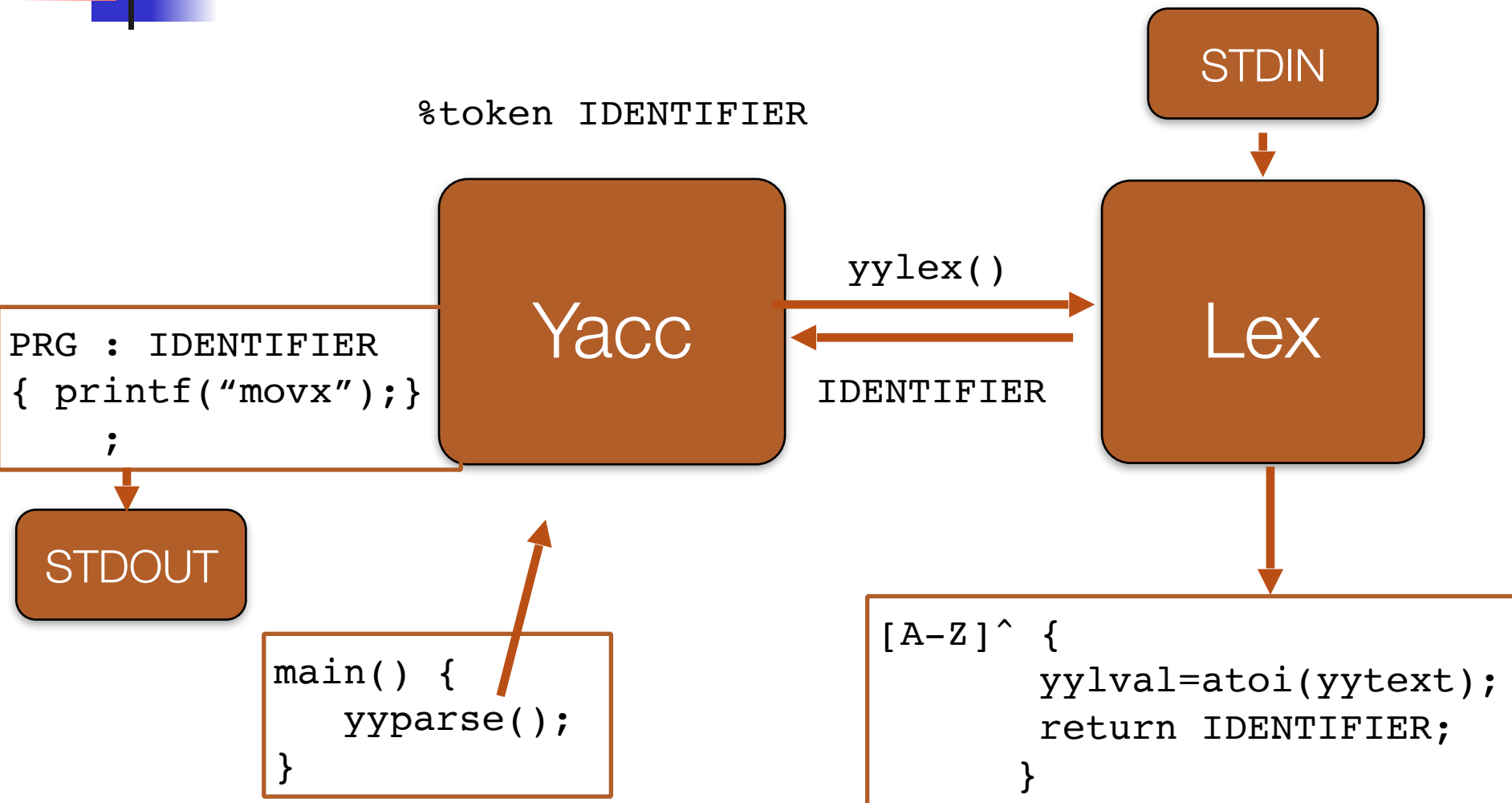


La película completa...

El código que se construye en los snippets de las acciones de parser.y permite generar un PROGRAMA (el compilador) que va a generar programas que pueden a su vez compilarse y generar un binario para alguna plataforma.



La película completa...





Tips

```
gcc -S snippet.c -o snipper.asm  
gcc -c snippet.asm -o program
```

AST: Abstract Syntax Tree : Incluyan una estructura de árbol que puedan utilizar para representar los nodos de algún programa que construyan y que además se pueda visualizar.

Aunque los errores de shift/reduce o reduce/reduce se informan como warnings, implica que internamente yacc tomó una decisión para solucionar el conflicto y la gramática que termina generando puede no ser exactamente como ustedes la pensaron.



Ejemplos

<https://github.com/faturita/LlvmBasicCompiler>

<https://github.com/faturita/YetAnotherCompilerClass>



Referencias

- Aho, Ullman's Dragoon Book.
- Cooper, Engineering a Compiler.
- <http://dinosaur.compilertools.net/yacc/>
- Scott, Programming Languages Pragmatic, 2009
- Harper, Practical Foundations for Programming Languages, 2013
- Brown, Levin, Mason, Lex & Yacc, 1998
- Paul Carter, PC Assembly Language, 2006