Lex en YACC inleiding/HOWTO

PowerDNS BV (bert hubert <bert at powerdns.com>),
vertaald door Paul Boekholt (p.boekholt at hetnet.nl)
v0.8 $Date: 2003/07/07 18:37:39 $

Dit document dient om je op weg te helpen met het gebruik van Lex en YACC

Inhoudsopgave

1 Inleiding

Welkom beste lezer.

Als je enige tijd in een Unix omgeving hebt geprogrammeerd, ben je vast de mystieke programma's Lex & YACC, of Flex & Bison, zoals ze wereldwijd genoemd worden door GNU/Linux gebruikers, tegengekomen, Flex zijnde een implementatie van Lex door Vern Paxon en Bison zijnde de GNU versie van YACC. We zullen deze programma's verder Lex en YACC noemen - de nieuwere versies zijn opwaarts compatible, dus je kunt Flex en Bison gebruiken bij het proberen van onze programma's.

Deze programma's zijn waanzinnig nuttig, maar net als bij je C compiler legt de manpage de taal die ze verstaan niet uit, noch hoe ze te gebruiken. YACC is echt fantastisch wanneer gebruikt met Lex, echter, de Bison manpage beschrijft niet hoe je door Lex gegenereerde code integreert met je Bison programma.

1.1 Wat dit document NIET is

Er zijn geweldige boeken over Lex & YACC. Lees in ieder geval deze boeken als je meer wilt weten. Ze geven veel meer informatie dan wij ooit kunnen. Zie de sectie "Lees Verderäan het eind. Dit document is bedoeld om je op weg te helpen bij het gebruik van Lex & YACC, om je in staat te stellen je eerste programma's te maken.

De documentatie die bij Flex en Bison hoort is ook uitstekend, maar geen leerboek. Maar ze vormen een aardige aanvulling op mijn HOWTO. Zie de verwijzingen aan het einde.

Ik ben zeker geen YACC/Lex expert. Toen ik begon dit document te schrijven, had ik precies twee dagen ervaring. Mijn enige doel is die twee dagen makkelijker voor je te maken.

Verwacht niet de juiste YACC en Lex stijl in deze HOWTO aan te treffen. Voorbeelden zijn uiterst eenvoudig gehouden en er zijn misschien betere manieren om ze te schrijven. Als je weet hoe, laat het me dan weten.

1.2 Spul downloaden

Merk op dat je al de gegeven voorbeelden kunt downloaden in machine-leesbare vorm. Zie de *homepage* `<http://ds9a.nl/lex-yacc>`voor details.

1.3 Licentie

2 Wat Lex & YACC voor je kunnen doen

Bij juist gebruik maken deze programma's het mogelijk met gemak complexe talen te scannen. Dit is een groot voordeel als je een configuratiebestand wilt lezen, of een compiler wilt schrijven voor welke taal dan ook die je (of iemand anders) misschien hebt uitgevonden.

Met een beetje hulp, die dit document hopelijk biedt, zul je nooit meer met de hand een parser hoeven te schrijven - Lex & YACC zijn de gereedschappen om dit te doen.

2.1 Wat ieder programma op zichzelf doet

Hoewel deze programma's uitblinken wanneer ze samen gebruikt worden, hebben ze ieder een eigen doel. Het volgende hoofdstuk legt uit wat ieder onderdeel doet.

3 Lex

Het programma Lex genereert een zogenaamde 'Lexer'. Dit is een functie die een stroom karakters als input neemt, en steeds als het een groep karakters ziet die met een sleutelwaarde overeenkomen, een bepaalde actie onderneemt. Een heel eenvoudig voorbeeld:

```
%{
#include <stdio.h>
%}

%%
stop    printf("Stop commando ontvangen\n");
start   printf("Start commando ontvangen\n");
%%
```

De eerste sectie, tussen de %{ en %} wordt direct overgenomen in het gegenereerde programma. Dit is nodig omdat we later printf gebruiken, wat gedefinieerd is in stdio.h.

Secties worden gescheiden door '%%', dus de eerste regel van de tweede sectie start met de 'stop' sleutel. Iedere keer dat de 'stop' sleutel tegengekomen wordt in de input, wordt de rest van de regel (een print() call) uitgevoerd.

Behalve 'stop' hebben we ook een 'start' gedefinieerd, die verder grotendeels hetzelfde doet.

We beëindigen de code weer met '%%'.

Doe dit om Voorbeeld 1 te compileren:

```
lex example1.l
cc lex.yy.c -o example1 -ll

OPMERKING:Als je flex gebruikt ipv lex, moet je misschien '-ll' vervangen door '-lfl' in
de compilatiescripts.  Redhat 6.x en SuSE vereisen dit, zelfs als je 'flex' aanroept als
'lex'!
```

Dit genereert het programma 'example1'. Als je het draait, wacht het tot je iets intikt. Als je iets typt dat niet overeenkomt met een gedefinieerde sleutel ('stop' en 'start') wordt het weer geoutput. Als je 'stop' intikt geeft het 'stop commando ontvangen';

Sluit af met een EOF (^D).

Je vraagt je misschien af hoe het programma draait, daar we geen main() gedefinieerd hebben. Die functie is voor je gedefinieerd in libl (liblex) die we ingecompileerd hebben met het -ll commando.

3.1 Reguliere expressies in matches

Dit voorbeeld was op zichzelf niet erg nuttig, en dat is het volgende ook niet. Het laat echter wel zien hoe je reguliere expressies gebruikt in Lex, wat later superhandig zal zijn.

Voorbeeld 2:

```
%{
#include <stdio.h>
%}

%%
[0123456789]+           printf("NUMMER\n");
[a-zA-Z][a-zA-Z0-9]*    printf("WOORD\n");
%%
```

Dit Lex bestand beschrijft twee soorten matches (tokens): WOORDen en NUMMERs. Reguliere expressies kunnen behoorlijk intimiderend zijn maar met een beetje werk zijn ze makkelijk te begrijpen. Laten we de NUMMER match bekijken:

[0123456789]+

Dit betekent: een reeks van een of meer karakters uit de groep 0123456789. We hadden het ook kunnen afkorten tot:

[0-9]+

De WOORD match is iets ingewikkelder:

[a-zA-Z][a-zA-Z0-9]*

Het eerste deel matcht 1 en slechts 1 karakter tussen 'a' en 'z', of tussen 'A' en 'Z'. Met andere woorden, een letter. Deze beginletter moet gevolgd worden door nul of meer karakters die ofwel een letter of een cijfer zijn. Waarom hier een asterisk gebruiken? De '+' betekent 1 of meer matches, maar een WOORD kan heel goed uit slechts 1 karakter bestaan, dat we al gematcht hebben. Dus het tweede deel heeft misschien nul matches, dus schrijven we een '*'.

Op deze manier hebben we het gedrag van veel programmeertalen geïmiteerd die vereisen dat een variabelenaam met een letter *moet* beginnen maar daarna cijfers mag bevatten. Met andere woorden, 'temperature1' is een geldige naam, maar '1temperature' niet.

Probeer voorbeeld 2 te compileren, net zoals voorbeeld 1, en voer wat tekst in. Een voorbeeldsessie:

```
<tscreen><verb>
$ ./example2
foo
WOORD

bar
WOORD

123
NUMMER

bar123
WOORD

123bar
```

```
NUMMER
WOORD
```

Je vraagt je misschien ook af waar al die witregels in de uitvoer vandaan komen. De reden is eenvoudig: het was in de invoer, en het wordt nergens gematcht, dus wordt het weer uitgevoerd.

De Flex manpage beschrijft de reguliere expressies in detail. Velen vinden de perl reguliere expressies manpage (perlre) ook nuttig, al kan Flex niet alles dat perl kan.

Zorg dat je geen matches met een lengte van nul zoals '[0-9]*' maakt - je lexer kan in de war raken en lege strings herhaaldelijk matchen.

3.2 Een ingewikkelder voorbeeld voor een C-achtige syntax

Laten we zeggen dat we een bestand willen parsen dat er zo uitziet:

```
logging {
        category lame-servers { null; };
        category cname { null; };
};

zone "." {
        type hint;
        file "/etc/bind/db.root";
};
```

We herkennen duidelijk een aantal categorieën (tokens) in dit bestand:

- WOORDen, zoals 'zone' en 'type'
- BESTANDSNAAMen, zoals 'etc/bind/db.root'
- QUOTEs, zoals om de bestandsnaam
- OBRACEs, {
- EBRACEs, }
- PUNTKOMMAs, ;

Het overeenkomstige Lex bestand is Voorbeeld 3:

```
%{
#include <stdio.h>
%}

%%
[a-zA-Z][a-zA-Z0-9]*    printf("WOORD ");
[a-zA-Z0-9\/.-]+        printf("BESTANDSNAAM ");
\"                      printf("QUOTE ");
\{                      printf("OBRACE ");
\}                      printf("EBRACE ");
;                       printf("PUNTKOMMA ");
\n                      printf("\n");
[ \t]+                  /* negeer whitespace */;
%%
```

Als we ons bestand aan het programma toevoeren dat door dit Lex bestand gegenereerd is (met gebruik van example3.compile) krijgen we:

```
WOORD OBRACE
WOORD BESTANDSNAAM OBRACE WOORD PUNTKOMMA EBRACE PUNTKOMMA
WOORD WOORD OBRACE WOORD PUNTKOMMA EBRACE PUNTKOMMA
EBRACE PUNTKOMMA

WOORD QUOTE BESTANDSNAAM QUOTE OBRACE
WOORD WOORD PUNTKOMMA
WOORD QUOTE BESTANDSNAAM QUOTE PUNTKOMMA
EBRACE PUNTKOMMA
```

Als we dit vergelijken met het bovengenoemde configuratiebestand, wordt duidelijk dat we het netjes 'getokeniseerd' hebben. Ieder deel van het configuratiebestand is gematcht, en omgezet naar een token.

3.3 Wat we geleerd hebben

We hebben geleerd dat Lex in staat is om willekeurige invoer te lezen, en te bepalen wat ieder onderdeel van de invoer is. Dit wordt 'tokenizering' genoemd.

4 YACC

YACC kan invoerstromen parsen die bestaan uit tokens met bepaalde waarden. Dit geeft precies weer hoe YACC zich verhoudt tot LEX, YACC weet niet wat 'invoerstromen' zijn, het heeft voorbewerkte tokens nodig. Je kunt je eigen tokenizeerder schrijven, maar we laten dit aan Lex over.

Een opmerking over grammatica's en parsers. Toen YACC het levenslicht zag, werd het gebruikt om invoerbestanden voor compilers te parsen: programma's. Programma's in een computertaal zijn in het algemeen *niet* dubbelzinnig - ze hebben maar een betekenis. YACC kan dus niet met dubbelzinnigheid omgaan en zal klagen over shift/reduce en reduce/reduce conflicten. Meer over dubbelzinnigheid en YACC "problemenïn het hoofdstuk 'Conflicten'.

4.1 Een eenvoudige thermostaat beheerser

Laten we zeggen dat we een thermostaat willen beheersen met een eenvoudige taal. Een sessie met de thermostaat kan er als volgt uit zien:

```
warmte aan
        Verwarming aan!
warmte uit
        Verwarming uit!
```

De tokens die we moeten herkennen zijn: warmte, aan/uit (TOESTAND), doel, temperatuur, NUMMER

De Lex tokenizeerder (Voorbeeld 4) is:

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
```

```
%%
[0-9]+                  return NUMMER;
warmte                  return TOKWARMTE;
aan|uit                 return TOESTAND;
doel                    return TOKDOEL;
temperatuur             return TOKTEMPERATUUR;
\n                      /* negeer einde regel */;
[ \t]+                  /* negeer whitespace */;
%%
```

We merken twee belangrijke veranderingen op. Ten eerste includen we het bestand 'y.tab.h' en ten tweede printen we niet meer, we returnen namen van tokens. Deze verandering is omdat we het nu allemaal aan YACC toevoeren, die is niet geïnteresseerd in wat we op het scherm uitvoeren. Y.tab.h heeft definities voor deze tokens.

Maar waar komt y.tab.h vandaan? Het wordt gegenereerd door YACC vanuit het Grammatica Bestand dat we gaan creëren. Onze taal is eenvoudig dus de grammatica ook:

```
commands: /* leeg */
        | commands command
        ;

command:
        heat_switch
        |
        target_set
        ;

heat_switch:
        TOKWARMTE TOESTAND
        {
                printf("\tWarmte aan- of uitgezet\n");
        }
        ;

target_set:
        TOKDOEL TOKTEMPERATUUR NUMMER
        {
                printf("\tTemperatuur ingesteld\n");
        }
        ;
```

Het eerste gedeelte is wat ik de 'wortel' zal noemen. Het vertelt ons dat we 'commands' hebben, en dat deze commands bestaan uit individuele 'command' delen. Je ziet dat deze regel erg recursief is, want hij bevat weer het woord 'commands'. Dit betekent dat het programma nu een reeks commands een voor een kan reduceren. Zie het hoofdstuk 'Hoe werken Lex en YACC intern' voor belangrijke details over recursie.

De tweede regel definieert wat een command is. We ondersteunen maar twee soorten commands, de 'heat_switch' en de 'target_set'. Dat is de betekenis van het |-symbool - 'een command bestaat uit ofwel een heat_switch of een target_set'.

Een heat_switch bestaat uit het HEAT token, wat simpelweg het woord 'warmte' is gevolgd door een toestand (die we in het Lex bestand gedefinieerd hebben als 'aan' of 'uit').

Iets ingewikkelder is de target_set, die bestaat uit het TARGET token (het woord 'doel'), het TEMPERATURE token (het woord 'temperatuur') en een getal.

4.1.1 Een volledig YACC bestand

De vorige paragraaf toonde alleen het grammatica gedeelte van het YACC bestand, maar er is meer. Dit is de header die we hebben weggelaten:

```
%{
#include <stdio.h>
#include <string.h>

void yyerror(const char *str)
{
        fprintf(stderr,"fout: %s\n",str);
}

int yywrap()
{
        return 1;
}

main()
{
        yyparse();
}

%}

%token NUMMER TOKWARMTE TOESTAND TOKDOEL TOKTEMPERATUUR
```

(Noot van de vertaler: tussen header en grammatica moet nog een %% staan.) De functie yyerror() wordt door YACC aangeroepen als hij een fout tegenkomt. We voeren gewoon de doorgestuurde boodschap uit, maar er zijn slimmere dingen te doen. Zie de paragraaf 'Verder lezen' aan het einde.

De functie yywrap() kan gebruikt worden om verder te lezen uit een ander bestand. Het wordt bij EOF aangeroepen en dan kun je een ander bestand openen, en 0 returnen. Of je kunt 1 returnen, om aan te geven dat dit echt het einde is. Zie verder het hoofdstuk 'Hoe Lex en YACC intern werken'.

Dan is er nog de functie main(), die niets doet behalve alles in gang zetten.

De laatste regel definieert eenvoudig de tokens die we gaan gebruiken. Die worden uitgevoerd met y.tab.h als YACC is aangeroepen met de '-d' optie.

4.1.2 De thermostaat beheerser compileren & draaien

```
lex example4.l
yacc -d example4.y
cc lex.yy.c y.tab.c -o example4
```

Sommige dingen zijn veranderd. We roepen nu ook YACC aan om onze grammatica te compileren, wat y.tab.c en y.tab.h genereert. Dan roepen we als gewoonlijk Lex aan. Bij het compileren laten we de -ll vlag weg; we hebben nu onze eigen main() en hebben die van libl niet nodig.

```
OPMERKING: als je een foutmelding krijgt dat je compiler 'yylval' niet kan vinden, voeg
dan onder #include <y.tab.h> dit toe:

extern YYSSTYOE yylval;
```

```
Dit wordt uitgelegd in de paragraaf ‘Hoe Lex en YACC intern werken’.
```

Een voorbeeldsessie:

```
$ ./example4
warmte aan
        Warmte aan- of uitgezet
warmte uit
        Warmte aan- of uitgezet
doel temperatuur 10
        Temperatuur ingesteld
doel vochtigheid 20
fout: parse error
$
```

Dit is niet helemaal wat we wilden bereiken, maar om de leercurve behapbaar te houden kunnen we niet alles tegelijk presenteren.

4.2 De thermostaat uitbreiden om parameters te verwerken

Zoals we hebben gezien kunnen we de thermostaat commands nu correct parsen, en zelfs fouten correct opmerken. Maar als je misschien hebt geraden door de dubbelzinnige bewoordingen, heeft het programma geen idee wat het zou moeten doen, de waarden die je invoert worden er niet naar doorgestuurd.

Laten we beginnen met het vermogen om de nieuwe doeltemperatuur te lezen. Hiervoor moeten we de NUMMER match in de Lexer leren zichzelf in een integer waarde om te zetten, die dan ingelezen kan worden in YACC.

Als Lex een doel matcht, stopt het de tekst van de match in de string ‘yytext’. YACC op zijn beurt verwacht een waarde in ‘yylval’. In voorbeeld 5 zien we de voor de hand liggende oplossing:

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+                  yylval=atoi(yytext); return NUMMER;
warmte                  return TOKWARMTE;
aan|uit                 yylval=!strcmp(yytext,"aan"); return TOESTAND;
doel                    return TOKDOEL;
temperatuur             return TOKTEMPERATUUR;
\n                      /* negeer einde regel */;
[ \t]+                  /* negeer whitespace */;
%%
```

Zoals je ziet draaien we atoi() op yytext, en stoppen we de uitkomst in yylval, waar YACC het kan vinden. Net zoiets voor de TOESTAND match, waarbij we yylval op 1 zetten als deze ‘aan’ is. Merk op dat een aparte ‘aan’ en ‘uit’ match in Lex een sneller programma zou genereren, maar ik wilde voor de verandering een ingewikkelder regel en actie laten zien.

Nu moeten we YACC leren hoe hiermee om te gaan. Wat in Lex ‘yylval’ genoemd wordt, heeft in YACC een andere naam. Laten we de regel die het nieuwe temperatuurdoel instelt bekijken:

```
target_set:
```

```
        TOKDOEL TOKTEMPERATUUR NUMMER
        {
                printf("\tTemperatuur ingesteld op %d\n",$3);
        }
        ;
```

Om toegang te krijgen tot het derde gedeelte van de regel (NUMMER), moeten we $3 gebruiken. Steeds als yylex() returnt, wordt de waarde van yylval aan de terminal (vertaler: token) toegekend, en kan benaderd worden met de $-constructie.

Om hier verder op in te gaan, beschouw de nieuwe 'heat_switch' regel:

```
heat_switch:
        TOKWARMTE TOESTAND
        {
                if($2)
                        printf("\tWarmte aan\n");
                else
                        printf("\tWarmte uit\n");
        }
        ;
```

Als je nu example5 draait, voert het netjes uit wat je hebt ingevoerd.

4.3 Een configuratiebestand parsen

Laten we een deel van het eerder genoemde configuratiebestand herhalen:

```
zone "." {
        type hint;
        file "/etc/bind/db.root";
};
```

We hebben al een Lexer voor dit bestand. Nu hoeven we alleen nog maar een YACC grammatica te schrijven, en de Lexer aanpassen zodat het waarden teruggeeft in een vorm die YACC kan snappen.

In de lexer van Voorbeeld 6 zien we:

```
%{
#include <stdio.h>
#include "y.tab.h"
%}

%%

zone                    return ZONETOK;
file                    return FILETOK;
[a-zA-Z][a-zA-Z0-9]*    yylval=strdup(yytext); return WOORD;
[a-zA-Z0-9\/.-]+        yylval=strdup(yytext); return BESTANDSNAAM;
\"                      return QUOTE;
\{                      return OBRACE;
\}                      return EBRACE;
;                       return PUNTKOMMA;
\n                      /* negeer einde regel*/;
[ \t]+                  /* negeer whitespace */;
%%
```

Als je goed kijkt, zie je dat yylval veranderd is! We verwachten niet meer dat het een integer is, maar nemen aan dat het een char * is. Voor de eenvoud roepen we strdup aan en verspillen een hoop geheugen. Merk op dat dit geen probleem is in veel gevallen als je een bestand maar een keer hoeft te parsen, en dan exit.

We willen strings opslaan omdat we nu voornamelijk met namen te maken hebben: bestandsnamen en zonenamen. In een later hoofdstuk zullen we uitleggen hoe je met verschillende soorten data omgaat.

Om YACC over het nieuwe type van yylval te vertellen, voegen we dit toe aan de header van onze YACC grammatica:

```
#define YYSTYPE char *
```

De grammatica zelf is weer ingewikkelder. We hakken het in stukjes om het verteerbaarder te maken.

```
commands:
        |
        commands command PUNTKOMMA
        ;


command:
        zone_set
        ;

zone_set:
        ZONETOK quotedname zonecontent
        {
                printf("Complete zone for '%s' gevonden\n",$2);
        }
        ;
```

Dit is de intro, inclusief voornoemde recursieve 'wortel' Merk op dat we specificeren dat commands worden getermineerd (en gescheiden) door ;'s. We definiëren één soort command, de 'zone_set'. Deze bestaat uit het ZONE token (het woord 'zone'), gevolgd door een quotedname en de 'zonecontent'. De zonecontent begint eenvoudig:

```
zonecontent:
        OBRACE zonestatements EBRACE
```

Hij moet beginnen met een OBRACE, een {. Dan komen de zonestatements, gevolgd door een EBRACE, een }.

```
quotedname:
        QUOTE BESTANDSNAAM QUOTE
        {
                $$=$2;
        }
```

Deze sectie definieert een 'quotedname': een BESTANDSNAAM tussen QUOTES. Dan iets bijzonders: de waarde van een quotedname token is de waarde van de BESTANDSNAAM. Dit betekent dat de quotedname als waarde de bestandsnaam zonder quotes heeft.

Dat is wat het magische '$$=$2' doet. Het zegt: mijn waarde is de waarde van mijn tweede deel. Als nu naar de quotedname verwezen wordt in andere regels, en je benadert zijn waarde met de $-constructie, zie je de waarde die we hier ingesteld hebben met $$=$2.

```
OPMERKING: deze grammatica verslikt zich in bestandsnamen zonder een '.'  of een '/'
erin.

zonestatements:
        |
        zonestatements zonestatement PUNTKOMMA
        ;

zonestatement:
        statements
        |
        FILETOK quotedname
        {
                printf("Een zonefile naam '%s' tegengekomen\n", $2);
        }
        ;
```

Dit is een algemeen statement die alle soorten statements binnen het 'zone' blok afvangt. We zien opnieuw de recursiviteit.

```
block:
        OBRACE zonestatements EBRACE PUNTKOMMA
        ;

statements:
        | statements statement
        ;

statement: WOORD | block | quotedname
```

Dit definieert een blok, en 'statements' die er in gevonden kunnen worden.

Bij uitvoering ziet de uitvoer er zo uit:

```
$ ./example6
zone "." {
        type hint;
        file "/etc/bind/db.root";
        type hint;
};
Een zonefile naam '/etc/bind/db.root' tegengekomen
Complete zone for '.' gevonden
```

5 Een Parser in C++ maken

Hoewel Lex en YACC ouder zijn dan C++, is het mogelijk een C++ parser te maken. Flex heeft een optie om een C++ lexer te genereren, maar die zullen we niet gebruiken, want YACC kan er direct mee omgaan.

Ik geef er de voorkeur aan Lex een gewoon C bestand te laten genereren, en YACC C++ code te laten genereren. Als je dan je toepassing linkt, kom je misschien problemen tegen omdat de C++ code standaard geen C functies kan vinden, tenzij je het verteld hebt dat de functies extern "C"zijn.

Maak hiervoor een C header in YACC:

```
extern "C"
{
        int yyparse(void);
        int yylex(void);
        int yywrap()
        {
                return 1;
        }

}
```

Als je yydebug wilt declareren of veranderen, moet dat nu zo:

```
extern int yydebug;

main()
{
        yydebug=1;
        yyparse();
}
```

Dit is vanwege de Een Definitie Regel van C++, die geen meervoudige definities van yydebug toestaat.

Je zult misschien ook de #define van YYSTYPE in je Lex bestand moeten herhalen, vanwege C++ z'n strengere type checking.

Compileren gaat ongeveer zo:

```
lex bindconfig2.l
yacc --verbose --debug -d bindconfig2.y -o bindconfig2.cc
cc -c lex.yy.c -o lex.yy.o
c++ lex.yy.o bindconfig2.cc -o bindconfig2
```

Vanwege het -o statement heet y.tab.h nu bindconfig2.cc.h, dus hou daar rekening mee.

Samengevat: doe geen moeite om een Lexer in C++ te compileren, hou het in C. Maak je parser in C++ en leg je compiler uit dat sommige functies C functies zijn met extern "C"statements.

6 Hoe werken Lex en YACC intern

In het YACC bestand schrijf je je eigen main() functie, die op een gegeven moment yyparse() aanroept. De functie yyparse() is voor je aangemaakt door YACC, en komt in y.tab.c terecht.

yyparse() leest een stroom token/waarde paren van yylex(), welke beschikbaar gesteld moet worden. Je kunt deze functie zelf schrijven, of Lex dit laten doen. In onze voorbeelden hebben we dit aan Lex overgelaten.

De yylex() zoals geschreven door Lex leest karakters van een FILE * bestandspointer, yyin genaamd. Als je yyin niet instelt, is het de standard input. Hij voert uit naar yyout, als niet ingesteld is dat stdout. Je kunt yyin ook veranderen in de yywrap() functie die bij einde van een bestand wordt aangeroepen. Daarmee kun je een ander bestand openen, en verder parsen.

Als dat het geval is, moet hij 0 returnen. Als je na dit bestand wilt stoppen met parsen, moet hij 1 returnen.

Iedere aanroep van yylex() returnt een integer waarde die een token type voorstelt. Dit vertelt YACC wat voor token hij gelezen heeft. Optioneel mag het token een waarde hebben, die in de variabele yylval geplaatst moet worden.

Standaard is yylval van het type int, maar je kunt dit in het YACC bestand overriden door YYSTYPE te her#definen.

De Lexer moet toegang hebben tot yylval. Hiervoor moet deze gedeclareerd worden in de scope van de lexer als een externe variabele. De originele YACC doet dit niet voor je, dus moet je het volgende aan je lexer toevoegen, net onder de #include <y.tab.h>:

```
extern YYSTYPE yylval;
```

Bison, welke de meeste mensen tegenwoordig gebruiken, doet dit automatisch voor je.

6.1 Tokenwaarden

Zoals opgemerkt moet yylex() returnen wat voor soort token het is tegengekomen, en zijn waarde in yylval stoppen. Als deze tokens gedefinieerd zijn met het %token commando, worden er numerieke id's aan toegekend, beginnend bij 256.

Hierdoor is het mogelijk alle ascii karakters als token te gebruiken. Laten we zeggen dat je een rekenmachine aan het schrijven bent, tot nu toe zouden we de lexer als volgt hebben geschreven:

```
[0-9]+          yylval=atoi(yytext); return NUMMER;
[ \n]+          /* eet whitespace */;
-               return MINUS;
\*              return MULT;
\+              return PLUS;
...
```

Onze YACC grammatica zou dan het volgende bevatten:

```
exp:    NUMMER
        |
        exp PLUS exp
        |
        exp MINUS exp
        |
        exp MULT exp
```

Dit is onnodig ingewikkeld. Door karakters te gebruiken als afkortingen voor numerieke token id's kunnen we onze lexer herschrijven:

```
[0-9]+          yylval=atoi(yytext); return NUMMER;
[ \n]+          /* eet whitespace */;
.               return (int) yytext[0];
```

De laatste punt matcht alle verder niet gematchte karakters.

De YACC grammatica wordt dan:

```
exp:    NUMMER
        |
        exp '+' exp
        |
        exp '-' exp
        |
        exp '*' exp
```

Dit is veel korter en ook duidelijker. Je hoeft de ascii tokens nu niet met %token in de header te declareren, ze werken zo uit de doos.

Een ander goed ding van deze constructie is dat Lex nu alles zal matchen dat we het geven - wat het standaard gedrag om niet gematchte invoer naar stdout te echo'en vermijdt. Als een gebruiker van deze rekenmachine een ^ gebruikt, geeft het nu een parse error, in plaats van op stdout ge-echoot te worden.

6.2 Recursie: 'rechts is fout'

Recursie is een vitaal aspect van YACC. Zonder recursie kun je niet specificeren dat een bestand bestaat uit een reeks onafhankelijke commando's of statements. Van zichzelf is YACC alleen geïnteresseerd in de eerste regel, of de regel die je aanwijst als de startregel, met het '%start' symbool.

Recursie in YACC komt in twee smaken: rechts en links. Linker recursie, die je meestal moet gebruiken, ziet er zo uit:

```
commands: /* leeg */
        |
        commands command
```

Dit betekent: een commando is ofwel leeg, of het bestaat uit een of meer commando's, gevolgd door een commando. De manier waarop YACC werkt betekent dat je nu gemakkelijk individuele commando groepen kunt afhakken (aan de voorkant) en reduceren.

Vergelijk dit met rechter recursie, die er verwarrend genoeg voor velen beter uitziet:

```
commands: /* leeg */
        |
        command commands
```

Maar dit is kostbaar. Gebruikt als de %start regel, vereist het dat YACC alle commando's in je bestand op de stack houdt, wat een hoop geheugen kan gebruiken. Dus gebruik in ieder geval linker recursie voor het parsen van lange statements. Soms is het moeilijk rechter recursie te vermijden maar als je statements niet te lang zijn, hoef je je niet in bochten te wringen om linker recursie te gebruiken.

Als iets je commando's beëindigt (en dus scheidt), lijkt rechter recursie erg natuurlijk, maar is het nog steeds kostbaar:

```
commands: /* leeg */
        |
        command PUNTKOMMA commands
```

De juiste manier om dit te coderen is met linker recursie (ik heb het ook niet uitgevonden):

```
commands: /* leeg */
        |
        commands command PUNTKOMMA
```

Eerdere versies van deze HOWTO gebruikten abusievelijk rechter recursie. Met dank aan Markus Triska.

6.3 yylval voor gevorderden: %union

Momenteel moeten we *het* type van yylval definiëren. Dit is echter niet altijd toepasselijk. Er zijn tijden dat we meerdere datatypen moeten kunnen verwerken. Terugkerend naar onze hypothetische thermostaat, willen we misschien een verwarming die we willen beheersen uitkiezen, als volgt:

```
Verwarming hoofdgebouw
        'hoofdgebouw' verwarming geselecteerd
doel temperatuur 23
        'hoofdgebouw' verwarming doel temperatuur nu 23
```

Hiervoor moet yylval een union wezen, die zowel strings als integers kan bevatten - maar niet tegelijk.

Bedenk dat we YACC eerder verteld hebben welk type yylval was door YYSTYPE te definiëren. We zouden YYSTYPE op deze manier als union kunnen definiëren, maar YACC heeft hier een gemakkelijker methode voor: het %union statement.

Gebaseerd op Voorbeeld 4, schrijven we nu de YACC grammatica van Voorbeeld 7. Eerst de intro:

```
%token TOKVERWARMING TOKWARMTE TOKDOEL TOKTEMPERATUUR

%union
{
        int number;
        char *string;
}

%token <number> TOESTAND
%token <number> NUMMER
%token <string> WOORD
```

We definiëren onze union, die alleen een nummer en een string bevat. Dan leggen we YACC met een uitgebreide %token syntax uit welk deel van de union ieder token moet benaderen.

In dit geval laten we het TOESTAND token een integer gebruiken, net als zonet. Hetzelfde geldt voor het NUMMER token, dat we gebruiken om temperaturen te lezen.

Nieuw is het WOORD token, dat gedeclareerd is om een string te gebruiken.

Het lexerbestand verandert ook een beetje:

```
%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"
%}
%%
[0-9]+                  yylval.number=atoi(yytext); return NUMMER;
verwarming              return TOKVERWARMING;
warmte                  return TOKWARMTE;
aan|uit                 yylval.number=!strcmp(yytext,"aan"); return TOESTAND;
doel                    return TOKDOEL;
temperatuur             return TOKTEMPERATUUR;
[a-z0-9]+               yylval.string=strdup(yytext);return WOORD;
\n                      /* negeer einde regel */;
[ \t]+                  /* negeer whitespace  */;
%%
```

Zoals je ziet benaderen we de yylval niet meer direct, we voegen een suffix toe om aan te geven welk deel we willen benaderen. In de YACC grammatica hoeft dat echter niet, want YACC doet het voor ons:

```
heater_select:
        TOKVERWARMING WOORD
```

```
            {
                    printf("\tVerwarming '%s' geselecteerd\n",$2);
                    heater=$2;
            }
            ;
```

Vanwege de %token declaratie hierboven kiest YACC automatisch het 'string' lid uit onze union. Merk ook op dat we een kopie van $2 opslaan, die later gebruikt wordt om de gebruiker te vertellen naar welke verwarming hij commando's stuurt:

```
    target_set:
            TOKDOEL TOKTEMPERATUUR NUMMER
            {
                    printf("\tTemperatuur van verwarming '%s' ingesteld op %d\n",heater,$3);
            }
            ;
```

Lees voor meer details example7.y.

7 Debuggen

Het is belangrijk om debuggingsfaciliteiten te hebben, vooral als je aan het leren bent. Gelukkig kan YACC een hoop feedback geven. Deze feedback komt tegen de prijs van enige overhead, dus je moet enige switches geven om het aan te zetten.

Voeg als je je grammatica compileert, –debug en –verbose toe aan de YACC opdrachtregel. Voeg dit toe aan de C heading van je grammatica:

int yydebug = 1;

Dit zal het bestand 'y.output' genereren die de gemaakte toestandsmachine uitlegt.

Als je nu de gegenereerde binary draait, zal het een *hoop* uitvoer geven over wat er gebeurt, inclusief in welke toestand de toestandsmachine op dit moment is, en welke tokens er worden gelezen.

Peter Jinks schreef een pagina op *debugging* `<http://www.cs.man.ac.uk/~pjj/cs2121/debug.html>` die vaak voorkomende fouten en hun oplossingen bevat.

7.1 De toestandsmachine

Intern draait je YACC parser een zogenaamde 'toestandsmachine'. Zoals de naam zegt, is dit een machine die in verschillende toestanden kan verkeren. Dan zijn er regels die overgangen van de ene toestand naar de andere bepalen. Alles begint met de zogenaamde 'root' regel die ik eerder vermeld heb.

Citaat uit de uitvoer van y.output van Voorbeeld 7:

```
    state 0

        ZONETOK     , and go to state 1

        $default    reduce using rule 1 (commands)

        commands    go to state 29
        command     go to state 2
        zone_set    go to state 3
```

Standaard reduceert deze toestand met de 'commands' regel. Dit is de voornoemde recursieve regel die 'commands' definieert als zijnde opgebouwd uit individuele command statements, gevolgd door een puntkomma, gevolgd door mogelijk meer commands.

Deze toestand reduceert tot het iets tegenkomt dat het begrijpt, in dit geval een ZONETOK, d.i. het woord 'zone'. Dan gaat het naar toestand 1, die het zone command verder afhandelt:

```
state 1

    zone_set  ->  ZONETOK . quotedname zonecontent   (rule 4)

    QUOTE       , and go to state 4

    quotedname  go to state 5
```

De eerste regel heeft een '.' om aan te geven waar we zijn: we hebben net een ZONETOK gezien en zoeken nu naar een 'quotedname'. Blijkbaar begint een quotedname met een QUOTE, die ons naar toestand 4 stuurt.

Compileer om dit verder te volgen Voorbeeld 7 met de vlaggen uit de paragraaf Debuggen.

7.2 Conflicten: 'shift/reduce', 'reduce/reduce'

Steeds als YACC je waarschuwt over conflicten, kun je problemen verwachten.Het oplossen van deze conflicten schijnt iets van een kunstvorm te zijn die je een hoop over je taal kan leren. Misschien meer dan je wilde weten.

De problemen draaien om de vraag hoe een reeks tokens te interpreteren. Laten we aannemen dat we een taal hebben die deze commando's moet accepteren:

```
verwijder verwarming all
verwijder verwarming number1
```

Hiertoe definiëren we deze grammatica:

```
delete_heaters:
        TOKDELETE TOKVERWARMING mode
        {
                deleteheaters($3);
        }

mode:   WOORD

delete_a_heater:
        TOKDELETE TOKVERWARMING WOORD
        {
                delete($3);
        }
```

Je ruikt misschien al moeilijkheden. De toestandsmachine begint met het woord 'verwijder' te lezen, en moet dan op basis van het volgende token beslissen waar naartoe te gaan. Dit volgende token kan een mode zijn, die aangeeft hoe de verwarmingen te verwijderen, of de naam van een verwarming die te verwijderen is.

Het probleem is dat het volgende token in beide gevallen een WOORD is. YACC weet dus niet wat het moet doen. Dit leidt tot een 'reduce/reduce' waarschuwing, en een waarschuwing dat de 'delete_a_heater' node nooit bereikt zal worden.

In dit geval is het conflict gemakkelijk opgelost (door het eerste commando te hernoemen tot 'verwijder verwarmingen all' of door 'all' een apart token te maken), maar soms is het moeilijker. Het y.output bestand dat gegenereerd wordt als je YACC de –verbose vlag geeft kan enorm helpen.

8 Verder lezen

GNU YACC (Bison) komt met een heel aardig info-bestand (.info) dat de YACC syntax heel goed documenteert. Het vermeldt Lex maar één keer, maar verder is het heel goed. Je kunt .info-bestanden lezen met Emacs of met het heel aardige hulpmiddel 'pinfo'. Het is ook beschikbaar op de GNU site: *BISON Manual* `<http://www.gnu.org/manual/bison/>`

Flex komt met een goede manpage die erg nuttig is als je al een ruw begrip hebt van wat Flex doet. Het *Flex Manual* `<http://www.gnu.org/manual/flex/>` is ook online.

Na deze inleiding tot Lex en YACC kom je er misschien achter dat je meer informatie nodig hebt. Ik heb deze boeken nog niet gelezen, maar ze klinken goed:

Bison-The Yacc-Compatible Parser Generator

van Charles Donnelly and Richard Stallman. Een *Amazon* `<http://www.amazon.com/exec/obidos/ASIN/0595100325/qid=989165194/sr=1-2/ref=sc_b_3/002-7737249-1404015>` gebruiker vond het nuttig.

Lex & YACC

Van John R. Levine, Tony Mason and Doug Brown. Wordt beschouwd als het standaardwerk over dit onderwerp, maar is een beetje gedateerd. Besprekingen op *Amazon* `<http://www.amazon.com/exec/obidos/ASIN/1565920007/ref=sim_books/002-7737249-1404015>`

Compilers : Principles, Techniques, and Tools

Van Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Het 'Dragon Book'. Uit 1985 en ze blijven het maar herdrukken. Beschouwd als het standaardwerk over compiler constructie. *Amazon* `<http://www.amazon.com/exec/obidos/ASIN/0201100886/ref=sim_books/002-7737249-1404015>`

Thomas Niemann schreef een document over het schrijven van compilers en calculators met Lex & YACC. Je vindt het *hier* `<http://epaperpress.com/y_man.html>`

De gemodereerde usenet nieuwsgroep comp.compilers kan ook helpen maar hou in gedachten dat de mensen daar geen toegewijde parser helpdesk zijn! Lees voor je post hun interessante *pagina* `<http://compilers.iecc.com/>` en vooral de *FAQ* `<http://compilers.iecc.com/faq.txt>` Lex - A Lexical Analyzer Generator van M. E. Lesk and E. Schmidt is een van de originele naslagwerken. Je vindt het *hier* `<http://www.cs.utexas.edu/users/novak/lexpaper.htm>`

YACC: Yet Another Compiler-Compiler door Stephen C. Johnson is een van de originele naslagwerken. Je vindt het *hier* `<http://www.cs.utexas.edu/users/novak/yaccpaper.htm>` Het bevat nuttige wenken over stijl.

9 Dank aan

- Pete Jinks <pjj%cs.man.ac.uk>

- Chris Lattner <sabre%nondot.org>
- John W. Millaway <johnmillaway%yahoo.com>
- Martin Neitzel <neitzel%gaertner.de>
- Esmond Pitt <esmond.pitt%bigpond.com>
- Eric S. Raymond
- Bob Schmertz <schmertz%wam.umd.edu>
- Adam Sulmicki <adam%cfar.umd.edu>
- Markus Triska <triska%gmx.at>
- Erik Verbruggen <erik%road-warrior.cs.kun.nl>
- Gary V. Vaughan <gary%gnu.org> (lees zijn uitstekende *Autobook* `<http://sources.redhat.com/autobook>`)
- *Ivo van der Wijk* `<http://vanderwijk.info>` (*Amaze Internet* `<http://www.amaze.nl>`)