

Unix: narzędzia do tworzenia programów

Witold Paluszynski
witold.paluszynski@pwr.wroc.pl
<http://sequoia.ict.pwr.wroc.pl/~witold/>

Copyright © 2001–2005 Witold Paluszynski
All rights reserved.

Niniejszy dokument zawiera materiały do wykładu na temat narzędzi do tworzenia oprogramowania w systemie Unix. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

Autoconf — motywacja

- wiele wersji Unixa, zasadniczo podobnych, ale z czasem coraz więcej różnic; z punktu widzenia programisty były drobne różnice w zestawie systemowych plików nagłówkowych i funkcji systemowych, oraz drobne ale istotne różnice w działaniu niektórych podsystemów, np. drivera terminala i job control
 - ⇒ powodowało to trudności w kompilacji jednego pakietu źródłowego na różnych systemach
- standard POSIX miał te różnice wyeliminować, ale został przyjęty w różnym stopniu w różnych systemach, a także wprowadził nowe funkcje i mechanizmy, więc różnice nie znikły a wręcz się pogłębiły
 - ⇒ w efekcie napisanie programu z odpowiednią liczbą wariantów, aby uwzględnić różnice między różnymi systemami i ich wersjami było możliwe, używając mechanizmu `#ifdef`, lecz było nadzwyczaj skomplikowane

Autoconf — koncepcja działania

- zaczęto stosować narzędzia programowe do rozwiązania tego problemu; ich celem było automatyczne wygenerowanie przeprowadzających badania i próby docelowego systemu budując odpowiedni plik Makefile do kompilacji pakietu źródłowego
 - do tego używa się skryptu shellowego "configure" uruchamianego przez użytkownika przy kompilacji programu; można taki skrypt wygenerować automatycznie narzędziem autoconf; nieco inne podejście prezentuje imake, który jest programem binarnym generującym plik Makefile z wzorców
- obecnie narzędziem uniwersalnie stosowanym w procesie tworzenia i konfiguracji takich przenośnych pakietów jest autoconf:
 - autoconf tworzy skrypt configure, który dostosowuje pliki Makefile, nagłówkowe pliki konfiguracyjne, i inne wymagane pliki (z dostarczonych wzorców), do wymagań konkretnej platformy systemowej
 - automake tworzy wzorce Makefile ze specyfikacji wyższego rzędu Makefile.am

Model GNU dystrybucji i instalacji oprogramowania

rozpakowanie dystrybucji źródłowej pakietu:

```
% gunzip app1-1.0.tar.gz
% tar xf app1-1.0.tar
```

pliki dodatkowej dokumentacji: NEWS, README, AUTHORS, Changelog
konfiguracja i kompilacja:

```
% cd app1-1.0
% ./configure
% make
% make check
% ./app1
% make clean

# make install
# make uninstall
```

Przygotowanie dystrybucji oprogramowania

wygenerowanie niezbędnych plików:

```
% aclocal      # tworzy aclocal.m4
% autoconf     # tworzy skrypt configure z configure.ac i aclocal.m4
% touch NEWS README AUTHORS Changelog
% automake -a  # tworzy Makefile.in z Makefile.am
```

ewentualnie próba kompilacji:

```
% configure   # tworzy Makefile
% make        # kompiluje
```

przygotowanie pakietu do rozprowadzania:

```
% make dist   # tworzy dystrybucje app1-1.0.tar.gz
% make distcheck # to samo, plus sprawdza
```

automatyzacja powyższego procesu po wprowadzeniu zmian: autoreconf
(regeneracja nieinteligentna), lub *maintainer mode*

Przygotowanie danych wejściowych automake i autoconf

Makefile.am jest plikiem wejściowym dla automake'a i określa specyfikację projektu, z czego się składa i co ma być gdzie zainstalowane:

```
bin_PROGRAMS = app1
app1_SOURCES = app1.c
```

Treść pliku określa, że będzie program 'app1' do zainstalowania w kartotece bin, i że jedynym plikiem źródłowym tego programu jest plik app1.c

configure.ac jest plikiem wejściowym autoconf'a i służy do wyprodukowania skryptu configure:

```
AC_INIT(app1.c)
AM_INIT_AUTOMAKE(app1, 1.0)
AC_PROG_CC
AC_PROG_INSTALL
AC_OUTPUT(Makefile)
```

Ten plik dokonuje niezbędnej inicjalizacji autoconf'a i automake'a, konfiguruje użycie kompilatora C i programu instalacji oprogramowania, oraz określa, że należy wyprodukować plik Makefile (na podstawie pliku Makefile.in wygenerowanego przez automake z Makefile.am)

Autoconf — zasady ogólne

- generowany plik *configure* powinien określić środowisko systemowe, w którym kompilowany jest program, i wygenerować odpowiednie definicje i opcje kompilacji
- techniczne elementy wywołania makr `autoconf`'a można znaleźć w instrukcji i nie stanowi to problemu:
 - `autoconf` definiuje makra sprawdzające dla typowych elementów, i instrukcja określa co daje wywołanie odpowiedniego makra
 - w przypadku braku gotowego testu na element, jakiego potrzebujemy można użyć testu bardziej ogólnego (np. na obecność pliku, gdy nie ma testu na obecność danego programu)
 - w ostateczności należy samemu napisać testujący fragment kodu w języku Bourne shella
- istotną i kluczową kwestią jest co należy sprawdzać: jedno możliwe podejście polega na inkrementalnej budowie pliku *configure.ac* i testowaniu wyników na wszystkich docelowych platformach systemowych
- optaca się utrzymanie się zasady maksymalnej przenośności, wyłączając być może systemy rzeczywiście archaiczne

Autoconf — przygotowanie `configure.ac`

typowa kolejność operacji w *configure.ac*:

1. inicjalizacja: typowe wywołania makr: `AC_INIT`, `AM_INIT_AUTOMAKE`, `AC_CONFIG_HEADER`, `AC_REVISION`
2. opcje: wywołania makr, które konfiguruja opcje komend
3. programy: sprawdzenie obecności programów niezbędnych w samym skrypcie *configure*, procesie kompilacji, albo potrzebnych dla pracy aplikacji
4. biblioteki: sprawdzenie obecności niezbędnych bibliotek, przez próbę linkowania z nimi
5. pliki nagłówkowe: również przez kompilację i linkowanie programów
6. struktury i typedef: korzystają ze znalezionych plików nagłówkowych
7. funkcje: sprawdzenie obecności funkcji typowo wymaga elementów wcześniej zlokalizowanych
8. generacja wyników: `AC_OUTPUT`

Autoconf — sprawdzanie programów

dobrą zasadą jest korzystanie wyłącznie z programów najbardziej rozpowszechnionych i tylko ze wspólnego podzbioru ich funkcjonalności w drugiej kolejności można korzystać z programów objętych normą POSIX, ewentualnie dodając testy sprawdzające krytyczne elementy na docelowych platformach

np. makro `AC_PROG_AWK` powoduje dodanie kodu dla sprawdzenia obecności programów: `mawk`, `gawk`, `nawk`, i `awk` (w tej kolejności), i ustawienie zmiennej `AWK` na pierwszy znaleziony program

inne makra sprawdzania programów o wielu alternatywnych wersjach: `AC_PROG_INSTALL`, `AC_PROG_YACC`, `AC_PROG_LEX`, `AC_PROG_LN_S`, `AC_PROG_RANLIB`

ogólne makra sprawdzania obecności programów: `AC_CHECK_PROG`, `AC_CHECK_TOOL`, `AC_PATH_PROG`, `AC_PATH_TOOL`

Autoconf — sprawdzanie bibliotek

różne Unixy mają różne biblioteki i nie ma zgodności jakie funkcje są w której bibliotece

Jednak sprawdzanie bibliotek nie jest trudne, polega na ich doborze i uwzględnieniu w pliku *Makefile*, i można nie przejmować się błędami w konfiguracji bibliotek, dopóki nie natrafimy na problem

korygowanie problemu z bibliotekami polega na znalezieniu odpowiedniej biblioteki w systemie, w którym wystąpił problem, i dodaniu testu na nią do *configure.ac*

Autoconf — „mały” przykład

struktura plików źródłowych:

- kartoteka główna: główne pliki organizacyjne pakietu autotools, takie jak 'configure', 'aclocal.m4', a także pliki informacyjne 'README' i inne
- biblioteki procedur wchodzące w skład projektu powinny mieć swoje podkartoteki z zestawem plików źródłowych oraz ich plikami 'Makefile.am'
- zestaw plików źródłowych programu głównego mogą być przechowywane w kartotece głównej projektu, jednak celowe jest, zwłaszcza w przypadku większych programów, ich oddelegowanie do podkartoteki tradycyjnie nazywanej 'src'
- dokumentacja w podkartotece 'doc'
- pakiet testów w podkartotece 'test'
- można również użyć podkartoteki, o nazwie np. 'config' do przechowywania plików roboczych pakietu autotools

Autoconf — sprawdzanie plików nagłówkowych

Przykład: makro *autoconf* *AC_HEADER_SYS_WAIT* sprawdza istnienie pliku nagłówkowego *sys/wait.h* i jego kompatybilność z normą *posix.1*; w takim przypadku *configure* definiuje makro preprocesora *HAVE_SYS_WAIT_H*.

```
#include <sys/types.h>
#if HAVE_SYS_WAIT_H
# include <sys/wait.h>
#endif

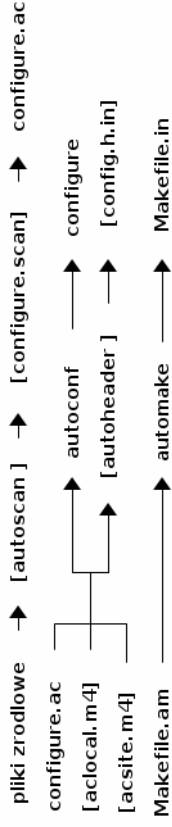
#ifdef WEXITSTATUS
# define WEXITSTATUS(stat_val) ((unsigned)(stat_val) >> 8)
#else
# define WEXITSTATUS(stat_val) (((stat_val) & 255) == 0)
#endif
```

Autoconf — przystosowanie istniejącego pakietu do autoconf

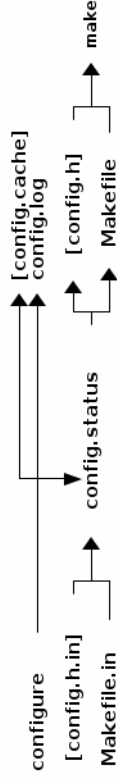
program *autotools* analizuje program źródłowy i generuje plik *configure.scan*, który ma być zgrubną przymiarką do *configure.ac*, ale który powinien być sprawdzony i poprawiony ręcznie

Zestaw narzędzi autoconf

Przygotowanie pakietu:



Instalacja pakietu:



Automake — wstęp

automake tworzy plik *Makefile.in*, który zawiera niezbędne reguły zależności pomiędzy elementami wchodzącymi w skład pakietu, jak również dodaje akcje pomocne w kompilacji, instalowaniu, testowaniu, usuwaniu, i ogólnie w utrzymywaniu pakietu oprogramowania w dobrym stanie.

podstawowe obiekty automake'a (*primaries*):

- DATA
- HEADERS
- SCRIPTS
- MANS
- TEXINFOS
- PROGRAMS
- LIBRARIES
- LTLIBRARIES

yacc — wstęp

```
%{
/* ewentualne konstrukcje C takie jak #include, #define, itp. */
%}
/* deklaracje yacc'a: tokeny, zmienne gramatyczne,
informacje o priorytetach i łączności konstrukcji gramatycznych */
%%
/* reguły gramatyczne i akcje */
%%
/* dalsze konstrukcje programowe w C */
main() { ...; yyparse(); ...; }
yylex() { ...; }
yyerror() { ...; }
```

yacc przekształca taką specyfikację do poniższej postaci, zastępując reguły gramatyczne deklaracją funkcji `yyparse()`, która, wywołana, dokona analizy gramatycznej ciągu wejściowego zgodnie z podanymi regułami.

```
konstrukcje C z pomiędzy %{ i %}
konstrukcje C z za drugiego %, to znaczy:
main() { ...; yyparse(); ...; }
yylex() { ...; }
yyerror() { ...; }
yyparse() { wygenerowany parser, który wywołuje yylex() }
```

Przykład: kalkulator czterodziałaniowy (hoc1)

Przedstawiony tu przykład kalkulatora został zaczerpnięty z książki „The UNIX Programming Environment”, autorzy: Brian Kernighan i Rob Pike, Prentice-Hall, 1984.

```
%{
#define YYSTYPE double /* typ danych stosu yacc'a */
%}
%token NUMBER
%left '+' '-' /* operatory lewostronnie łączne */
%left '*' '/' /* też lewostr.lączne, wyższy priorytet */
%%
list: /* nic */
| list '\n' { printf("\t%.8g\n", $2); }
;
expr: NUMBER { $$ = $1; }
| expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr { $$ = $1 / $3; }
| '(' expr ')' { $$ = $2; }
;
%%
```

W przedstawionej wersji kalkulatora zwróćmy uwagę na następujące szczegóły.

1. Reguły gramatyki mogą mieć dołączone akcje (w nawiasach klamrowych), które są wykonywane gdy parser dokonuje redukcji przez daną regułę, tzn. doszedł właśnie do wniosku, że rozpoznał konstrukcję odpowiadającą tej konkretnej regule w ciągu wejściowym.
2. Gramatyka definiuje listę wyrażeń jako główną konstrukcję. Reguły opisujące konstrukcję listę są rekurencyjne, tzn. dopuszczają wystąpienie wyrażenia po liście, bądź nawet samego znaku nowej linii po liście. Pozwała to na rozpoznanie dowolnej liczby wyrażeń (listy wyrażeń), a jednocześnie pustych linii nie zawierających wyrażenia. Pusta postać listy zapewnia inicjację, tzn. gwarantuje wystąpienie listy przed pierwszym wyrażeniem.
3. Fakt, że gramatyka sama zapewnia powtarzanie rozpoznawania wyrażenia jest kwestią wyboru. Reguły mogłyby być napisane tak aby rozpoznawać dokładnie jedno wyrażenie, a powtarzalność mogłaby zapewnić jawna pętla w programie głównym.

```
#include <stdio.h>
#include <ctype.h>
```

```
char *prognose; /* dla komunikatow o bledach */
int lineno = 1; /* liczenie wierszy wejścia */

main(int argc, char *argv[]) /* wersja 1 */
{
    progname = argv[0];
    yyparse();
}

yyerror(char *s) /* yacc wywołuje to gdy bład gramatyczny */
{
    warning(s, (char *) 0);
}

warning(char *s, char *t) /* wyświetla komunikat o bledzie */
{
    fprintf(stderr, "%s: %s", progname, s);
    if (t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
}
```

```
yylex()
{
    int c;
    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) { /* liczba */
        ungetc(c, stdin);
        scanf("%lf", &yyval);
        return NUMBER;
    }
    if (c == '\n')
        lineno++;
    return c; /* kazdy inny token */
}
```

hoc1.5: uwzględnienie minusa unarnego

```
%{
#define YYSTYPE double /* typ danych stosu yacc'a */
%}
%token
NUMBER
%left '+' '-' /* operatory lewostronnie laczne */
%left '*' '/' /* tez lewostr.laczne, wyzszy priorytet */
%left UNARYMINUS
%%
list: /* nic */
| list '\n'
| list expr '\n' { printf("\t%.8g\n", $2); }
;
expr: NUMBER { $$ = $1; }
| '-' expr %prec UNARYMINUS { $$ = -$2; }
| expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr { $$ = $1 / $3; }
| '(' expr ')' { $$ = $2; }
;
%%
```

1. Jednoznakowe tokeny pojawiające się w ciągu wejściowym występują w gramatyce w postaci samych znaków. Tokeny dłuższe niż jednoznakowe muszą być zadeklarowane jako %token, i występują w gramatyce w postaci odpowiednich makrodefinicji. Rozpoznawanie tych tokenów jest zadaniem funkcji yyLex() i gramatyka oczekuje jedynie informacji, że dany token wystąpił.
2. Deklaracje %left i %right określają tokeny łączne lewo-, lub prawostronnie. Istnieje również deklaracja %nonassoc określająca token nielączny. Kolejność występowania deklaracji %token, %left, %right i %nonassoc określa priorytet danego tokenu; późniejsze deklaracje określają wyższy priorytet.
3. W przypadku występowania jednego tokenu w kilku rolach, gdy ma on posiadać różne priorytety w zależności od roli (tak jak minus między wyrażeniami i UNARYMINUS przed wyrażeniem) możemy odstępstwo od normalnego priorytetu tokenu określić dodatkową deklaracją %prec umieszczaną na końcu reguły, w której token ma niestandardowy priorytet.

```

expr:
NUMBER
| VAR '=' expr { $$ = mem[$1]; }
| VAR '+' expr { $$ = mem[$1] + $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr {
if ($3 == 0.0)
execerror("division by zero", "");
$$ = $1 / $3; }
| '(' expr ')' { $$ = $2; }
| '-' expr %prec UNARYMINUS { $$ = -$2; }
;

%%

/* end of grammar */

#include <stdio.h>
#include <ctype.h>

char *progname;
int lineno = 1;

#include <signal.h>
#include <setjmp.h>
jmp_buf begin;

```

hoc2: zmienne i inne drobiazgi

```

%{
double mem[26]; /* memory for variables 'a'..'z' */
%}
%union {
double val; /* actual value */
int index; /* index into mem[] */
}

%token <val> NUMBER
%token <index> VAR
%type <val> expr
%right '+' '-'
%left '*' '/'
%left UNARYMINUS
%%

list: /* nothing */
| list '\n'
| list expr '\n' { printf("\t%.8g\n", $2); }
| list error '\n' { yerror; }
;

```

```

void fpeatch(int);

main(int argc, char *argv[]) { /* hoc2 */
progname = argv[0];
setjmp(begin);
signal(SIGFPE, fpeatch);
yyparse();
}

yylex() { /* hoc2 */
int c;

while ((c=getchar()) == ' ' || c == '\t')
;
if (c == EOF)
return 0;
if (c == '.' || isdigit(c)) { /* number */
ungetc(c, stdin);
scanf("%lf", &yyval.val);
return NUMBER;
}
if (islower(c)) {
yyval.index = c - 'a'; /* ASCII only */
return VAR;
}
}

```

```

extern double Pow();
%}
%union {
double val; /* actual value */
Symbol *sym; /* symbol table pointer */
}
%token <val> NUMBER
%token <sym> VAR BLTIN UNDEF
%type <val> expr asgn
%right '=',
%left '+', '-',
%left '*', '/',
%left UNARYMINUS
%right '~' /* exponentiation */
%%
list: /* nothing */
| list '\n',
| list asgn '\n',
| list expr '\n' { printf("\t%.8g\n", $2); }
| list error '\n' { yyerrok; }
;
asgn: VAR '=' expr { $$=$1->u.val=$3; $1->type = VAR; }
;
expr: NUMBER
| VAR { if ($1->type == UNDEF)

```

hoc3: funkcje matematyczne, lepsze zmienne, itp.

```

hoc.h:
=====
typedef struct Symbol { /* symbol table entry */
char *name;
short type; /* VAR, BLTIN, UNDEF */
union {
double val; /* if VAR */
double (*ptr)(); /* if BLTIN */
} u;
struct Symbol *next; /* to link to another */
} Symbol;
Symbol *lookup(char *s);
Symbol *install(char *s, int t, double d);

hoc.y:
=====
%{
#include "hoc.h"

```

```

execerror("undefined variable", $1->name);
$$ = $1->u.val; }
} asgn
| BLTIN '(' expr ')' { $$ = (*($1->u.ptr))($3); }
| expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr {
if ($3 == 0.0)
execerror("division by zero", "");
$$ = $1 / $3; }
| expr '~' expr { $$ = Pow($1, $3); }
| '(' expr ')' { $$ = $2; }
| '-' expr %prec UNARYMINUS { $$ = -$2; }
;
%%
/* end of grammar */
#include <stdio.h>
#include <ctype.h>
char *prognose;
int lineno = 1;
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;
void fpecatch(int);

```



```

}

warning(char *s, char *t) {
    fprintf(stderr, "%s: %s", progname, s);
    if (t && *t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
}

symbol.c:
=====
#include <stdlib.h>
#include "hoc.h"
#include "y.tab.h"

void *emalloc(unsigned n);
static Symbol *symlist = 0; /* symbol table: linked list */

Symbol *lookup(char *s) { /* find s in symbol table */
    Symbol *sp;
    for (sp = symlist; sp != (Symbol *) 0; sp = sp->next)

```

```

        if (strcmp(sp->name, s) == 0)
            return sp;
        /* 0 ==> not found */
    }

    Symbol *install(char *s, int t, double d){/* install s in symbol table */
        Symbol *sp;

        sp = (Symbol *) emalloc(sizeof(Symbol));
        sp->name = emalloc(strlen(s)+1); /* +1 for '\0' */
        strcpy(sp->name, s);
        sp->type = t;
        sp->u.val = d;
        sp->next = symlist; /* put at front of list */
        symlist = sp;
        return sp;
    }

    void *emalloc(unsigned n) { /* check return from malloc */
        void *p;
        p = malloc(n);
        if (p == 0)
            execerror("out of memory", (char *) 0);
        return p;
    }
}

```

```

main(int argc, char *argv[]) { /* hoc3 */
    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    yyparse();
}

yylex() {
    int c;
    while ((c=getchar()) == ' ' || c == '\t')
        ;
    if (c == EOF)
        return 0;
    if (c == '.' || isdigit(c)) { /* number */
        ungetc(c, stdin);
        scanf("%lf", &yyval.val);
        return NUMBER;
    }
    if (isalpha(c)) {
        Symbol *s;
        char sbuf[100], *p = sbuf;
        do {
            *p++ = c;
        } while ((c=getchar()) != EOF && isalnum(c));
        ungetc(c, stdin);
        *p = '\0';
        if ((s=lookup(sbuf)) == 0)
            s = install(sbuf, UNDEF, 0.0);
        yyval.sym = s;
        return s->type == UNDEF ? VAR : s->type;
    }
    if (c == '\n')
        lineno++;
    return c;
}

yyerror(char *s) {
    warning(s, (char *)0);
}

execerror(char *s, char *t) { /* recover from run-time error */
    warning(s, t);
    longjmp(begin, 0);
}

void fpecatch(int sig) { /* catch floating point exceptions */
    execerror("floating point exception", (char *) 0);
}

```

```

}

math.c:
=====
#include <math.h>
#include <errno.h>

extern int      errno;
double errcheck(double d, char *s);

double Log(double x) {
    return errcheck(log(x), "log");
}

double Log10(double x) {
    return errcheck(log10(x), "log10");
}

double Sqrt(double x) {
    return errcheck(sqrt(x), "sqrt");
}

}

init.c:
=====
#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double  Log(), Log10(), Exp(), Sqrt(), integer();

static struct {
    char *name;
    double cval;
} consts[] = {
    "PI",      3.14159265358979323846,
    "E",       2.71828182845904523536,
    "GAMMA",  0.57721566490153286060, /* Euler */
    "DEG",    57.29577951308232087680, /* deg/radian */
    "PHI",    1.61803398874989484820, /* golden ratio */
    0,        0
};

static struct {
    char *name;
    double (*func)(double);
} builtins[] = {

```

```

    "sin", sin,
    "cos", cos,
    "atan", atan,
    "log", Log, /* checks argument */
    "log10", Log10, /* checks argument */
    "exp", Exp, /* checks argument */
    "sqrt", Sqrt, /* checks argument */
    "int", integer,
    "abs", fabs,
    0, 0
};

double Exp(double x) {
    return errcheck(exp(x), "exp");
}

double Pow(double x, double y) {
    return errcheck(pow(x,y), "exponentiation");
}

double integer(double x) {
    return (double)(long)x;
}

double errcheck(double d, char *s) { /* check result of library call */
    if (errno == EDOM) {
        errno = 0;
        execerror(s, "argument out of domain");
    } else if (errno == ERANGE) {
        errno = 0;
        execerror(s, "result out of range");
    }
    return d;
}

```

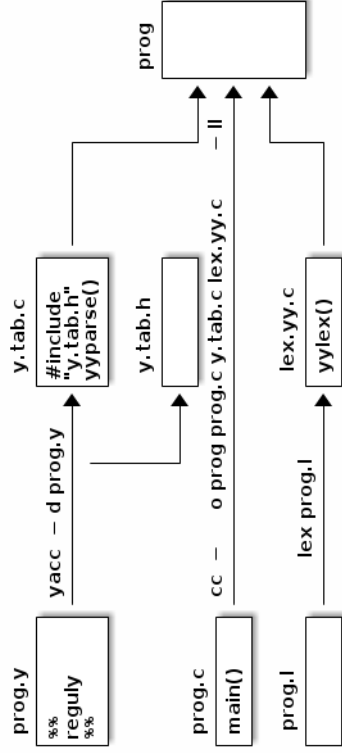
```

    "int", integer,
    "abs", fabs,
    0, 0
};

init() {
    int i;
    Symbol *s;
    for (i = 0; consts[i].name; i++)
        install(consts[i].name, VAR, consts[i].cval);
    for (i = 0; builtins[i].name; i++) {
        s = install(builtins[i].name, BLTIN, 0.0);
        s->u.ptr = builtins[i].func;
    }
}

```

Schemat zastosowania narzędzi yacc i lex do generacji programu ze źródeł typu .l i .y przy założeniu, że główna część programu znajduje się w pliku prog.c, specyfikacja gramatyki (wraz z niezbędnymi deklaracjami) w pliku prog.y, a specyfikacja tokenów w pliku prog.l.



hoc4: kompilacja kodu

przykład ze strony 258

```

#include "hoc.h"
#define code2(c1,c2) code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union {
    Symbol *sym; /* symbol table pointer */
    Inst *inst; /* machine instruction */
}
%token <sym> NUMBER VAR BLTIN UNDEF
%right '=',
%left '+', '-'
%left '*', '/',
%left UNARYMINUS
%right '^' /* exponentiation */
%%
list: /* nothing */
| list '\n'
| list asgn '\n' { code2(pop, STOP); return 1; }
| list expr '\n' { code2(print, STOP); return 1; }
| list error '\n' { yyerror; }
;
  
```

Użycie programu lex

```

%{
#include "y.tab.h"
extern int lineno;
%}
%%
[ \t] { ; } /* pomin spacje i taby */
[0-9]+\.[0-9]*|[0-9]+\.[0-9]+ {
    sscanf(yytext, "%lf", &yyval.val); return NUMBER; }
[a-zA-Z][a-zA-Z0-9]* {
    Symbol *s;
    if ((s=lookup(yytext)) == 0)
        s = install(yytext, UNDEF, 0.0);
    yyval.sym = s;
    return s->type == UNDEF ? VAR : s->type; }
\n
.
{ return yytext[0]; }
  
```

```

asgn: VAR '=' expr { code3(varpush, (Inst)$1, assign); }
;
expr: NUMBER
| VAR
| asgn
| BLTIN '(' expr ')' { code2(bltin, (Inst)$1->u.ptr); }
| '(' expr ')',
| expr '+' expr { code(add); }
| expr '-' expr { code(sub); }
| expr '*' expr { code(mul); }
| expr '/' expr { code(div); }
| expr '^' expr { code(power); }
| '-' expr %prec UNARYMINUS { code(negate); }
  
```

```

main(int argc, char *argv[]) { /* hoc4 */
  int fpecatch();

  progname = argv[0];
  init();
  setjmp(begin);
  signal(SIGFPE, fpecatch);
  for (initcode(); yyparse(); initcode())
    execute(prog);
  return 0;
}

static int c; /* globalna do uzycia w warning() */

yylex() { /* hoc4 */
  ...
  if (c == '.' || isdigit(c)) { /* liczba */
    double d;
    ungetc(c, stdin);
    scanf("%lf", &d);
    yylval.sym = install("", NUMBER, d);
    return NUMBER;
  }
  ... }
}

#include "hoc.h"
#include "y.tab.h"

#define NSTACK 256
static Datum stack[NSTACK]; /* the stack */
static Datum *stackp; /* next free spot on stack */

#define NPROG 2000
Inst prog[NPROG]; /* the machine */
Inst *progp; /* next free spot for code generation*/
Inst *pc; /* program counter during execution */

initcode() { /* initialize for code generation */
  stackp = stack;
  progp = prog;
}

push(Datum d) { /* push d onto stack */
  if (stackp >= &stack[NSTACK])
    execeerror("stack overflow", (char *) 0);
  *stackp++ = d;
}

Datum pop() { /* pop and return top elem from stack */
  if (stackp <= stack)

```

```

typedef struct Symbol { /* symbol table entry */
  char *name;
  short type; /* VAR, BLTIN, UNDEF */
  union {
    double val; /* if VAR */
    double (*ptr)(); /* if BLTIN */
  } u;
  struct Symbol *next; /* to link to another */
} Symbol;
Symbol *install(), *lookup();

typedef union Datum { /* interpreter stack type */
  double val;
  Symbol *sym;
} Datum;
extern Datum pop();

typedef int (*Inst)(); /* machine instruction */
#define STOP (Inst) 0

extern Inst prog[];
extern eval(), add(), sub(), mul(), div(),
extern negate(), power(), assign(), bltin();
extern varpush(), constpush(), print();

```

```

    execeerror("stack underflow", (char *) 0);
  return *--stackp;
}

```

```

Inst *code(Inst f) { /* install one instruction or operand */
  Inst *oprogp = progp;
  if (progp >= &prog[NPROG])
    execerror("program too big", (char *) 0);
  *progp++ = f;
  return oprog;
}

execute(Inst *p) { /* run the machine */
  for (pc = p; *pc != STOP; )
    (*(pc++))();
}

constpush() { /* push constant onto stack */
  Datum d;
  d.val = ((Symbol *)pc++)->u.val;
  push(d);
}

varpush() { /* push variable onto stack */
  Datum d;
  d.sym = (Symbol *)pc++;
}

```

```

add() { /* add top two elems on stack */
  Datum d1, d2;
  d2 = pop();
  d1 = pop();
  d1.val += d2.val;
  push(d1);
}

sub() { /* subtract top of stack from next */
  Datum d1, d2;
  d2 = pop();
  d1 = pop();
  d1.val -= d2.val;
  push(d1);
}

mul() {
  Datum d1, d2;
  d2 = pop();
  d1 = pop();
  d1.val *= d2.val;
  push(d1);
}

div() {

```

```

  push(d);
}

```

```

  Datum d1, d2;
  d2 = pop();
  if (d2.val == 0.0)
    execerror("division by zero", (char *) 0);
  d1 = pop();
  d1.val /= d2.val;
  push(d1);
}

negate() {
  Datum d;
  d = pop();
  d.val = -d.val;
  push(d);
}

power() {
  Datum d1, d2;
  extern double Pow();
  d2 = pop();
  d1 = pop();
  d1.val = Pow(d1.val, d2.val);
  push(d1);
}

```

hoc5: operatory relacyjne i instrukcje złożone

```
eval() {
    Datum d;
    d = pop();
    if (d.sym->type == UNDEF)
        execerror("undefined variable", d.sym->name);
    d.val = d.sym->u.val;
    push(d);
}

assign() {
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
            d1.sym->name);
    d1.sym->u.val = d2.val;
    d1.sym->type = VAR;
    push(d2);
}

print() {
    Datum d;
    d = pop();
    printf("\t%.8g\n", d.val);
}
```

```
%{
#include "hoc.h"
#define code2(c1,c2) code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union {
    Symbol *sym; /* symbol table pointer */
    Inst *inst; /* machine instruction */
}
%token <sym> NUMBER PRINT VAR BLTIN UNDEF WHILE IF ELSE
%type <inst> stmt asgn expr stmtlist cond while if end
%right '='
%left OR
%left AND
%left GT GE LT LE EQ NE
%left '+' '-'
%left '*' '/'
%left UNARYMINUS NOT
%right '^'
%%
list: /* nothing */
    | list '\n'
```

```
}

bltin() {
    Datum d;
    d = pop();
    d.val = (*(double (*)())(*pc++))(d.val);
    push(d);
}
```

```
| list asgn '\n' { code2(pop, STOP); return 1; }
| list stmt '\n' { code(STOP); return 1; }
| list expr '\n' { code2(print, STOP); return 1; }
| list error '\n' { yyerror; }
;
VAR '=' expr { $$=$3; code3(varpush,(Inst)$1,assign); }
;
stmt:
    expr { code(pop); }
| PRINT expr { code(prexpr); $$ = $2; }
| while cond stmt end {
    ($1)[1] = (Inst)$3; /* body of loop */
    ($1)[2] = (Inst)$4; } /* end, if cond fails */
| if cond stmt end { /* else-less if */
    ($1)[1] = (Inst)$3; /* thenpart */
    ($1)[3] = (Inst)$4; } /* end, if cond fails */
| if cond stmt end ELSE stmt end { /* if with else */
    ($1)[1] = (Inst)$3; /* thenpart */
    ($1)[2] = (Inst)$6; /* elsepart */
    ($1)[3] = (Inst)$7; } /* end, if cond fails */
| '{' stmtlist ',' { $$ = $2; }
;
'(' expr ')' { code(STOP); $$ = $2; }
;
while: { $$ = code3(whilecode, STOP, STOP); }
;
```

```

if: IF { $$=code(ifcode); code3(STOP, STOP, STOP); }
;
end: /* nothing */ { code(STOP); $$ = prog; }
;
stmtlist: /* nothing */ { $$ = prog; }
| stmtlist '\n'
| stmtlist stmt
;
expr: NUMBER { $$ = code2(constpush, (Inst)$1); }
| VAR { $$ = code3(varpush, (Inst)$1, eval); }
| asgn
| BLTIN (' expr ')
| '(' expr ')' { $$ = $2; }
| expr '+' expr { code(add); }
| expr '-' expr { code(sub); }
| expr '*' expr { code(mul); }
| expr '/' expr { code(div); }
| expr '^' expr { code(power); }
| '-' expr %prec UNARYMINUS { $$ = $2; code(negate); }
| expr GT expr { code(gt); }
| expr GE expr { code(ge); }
| expr LT expr { code(lt); }
| expr LE expr { code(le); }
| expr EQ expr { code(eq); }
;

```

```

ungetc(c, stdin);
scanf("%lf", &d);
yyval.sym = install("", NUMBER, d);
return NUMBER;
}
if (isalpha(c)) {
Symbol *s;
char sbuf[100], *p = sbuf;
do
    *p++ = c;
while ((c=getchar()) != EOF && isalnum(c));
ungetc(c, stdin);
*p = '\0';
if ((s=lookup(sbuf)) == 0)
    s = install(sbuf, UNDEF, 0.0);
yyval.sym = s;
return s->type == UNDEF ? VAR : s->type;
}
switch (c) {
case '>': return follow('=', GE, GT);
case '<': return follow('=', LE, LT);
case '=': return follow('=', EQ, '=');
case '!': return follow('=', NE, NOT);
case '|': return follow('|', OR, '|');
case '&': return follow('&', AND, '&');
}

```

```

case '\n': lineno++; return '\n';
default: return c;
}
follow(expect, ifyes, ifno) /* look ahead for >=, etc. */
{
int c = getchar();
if (c == expect)
    return ifyes;
ungetc(c, stdin);
return ifno;
}
yyerror(char *s) {
warning(s, (char *)0);
}
execerror(char *s, char *t) /* recover from run-time error */
{
warning(s, t);
longjmp(begin, 0);
}
fpcatch() /* catch floating point exceptions */
{
execerror("floating point exception", (char *) 0);
}

```

```

| expr NE expr { code(ne); }
| expr AND expr { code(and); }
| expr OR expr { code(or); }
| NOT expr { $$ = $2; code(not); }
;
/* end of grammar */
#include <stdio.h>
#include <ctype.h>
char *progname;
int lineno = 1;
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;
int defining;
int c; /* global for use by warning() */
yylex() /* hoc5 */
{
while ((c=getchar()) == ' ' || c == '\t')
;
if (c == EOF)
return 0;
if (c == '.' || isdigit(c)) /* number */
double d;
}

```

```

extern ifcode(), whilecode();

}

main(int argc, char *argv[]) {
    int fpecatch();

    progname = argv[0];
    init();
    setjmp(begin);
    signal(SIGFPE, fpecatch);
    for (initcode(); Yyparse(); initcode())
        execute(prog);
    return 0;
}

warning(char *s, char *t) {
    fprintf(stderr, "%s: %s", progname, s);
    if (t && *t)
        fprintf(stderr, " %s", t);
    fprintf(stderr, " near line %d\n", lineno);
    while (c != '\n' && c != EOF)
        c = getchar(); /* flush rest of input line */
    fseek(stdin, 0L, 2); /* flush rest of file */
    longjmp(begin, 0);
}

```

```

typedef struct Symbol { /* symbol table entry */
    char *name;
    short type; /* VAR, BLTIN, UNDEF */
    union {
        double val; /* if VAR */
        double (*ptr)(); /* if BLTIN */
    } u;
    struct Symbol *next; /* to link to another */
} Symbol;
Symbol *install(), *lookup();

typedef union Datum { /* interpreter stack type */
    double val;
    Symbol *sym;
} Datum;
extern Datum pop();

typedef int (*Inst)(); /* machine instruction */
#define STOP (Inst) 0

extern Inst prog[], *progp, *code();
extern eval(), add(), sub(), mul(), div(), negate(), power();
extern assign(), bltin(), varpush(), constpush(), print();
extern prexpr();
extern gt(), lt(), eq(), ge(), le(), ne(), and(), or(), not();

```

```

#include "hoc.h"
#include "y.tab.h"

#define NSTACK 256

static Datum stack[NSTACK];
static Datum *stackp;

#define NPROG 2000
Inst prog[NPROG];
static Inst *pc;
Inst *progp;

initcode()
{
    progp = prog;
    stackp = stack;
}

push(d)
Datum d;
{
    if (stackp >= &stack[NSTACK])
        execerror("stack too deep", (char *)0);
    *stackp++ = d;
}

```



```

}
Datum
pop()
{
    if (stackp == stack)
        execerror("stack underflow", (char *)0);
    return *--stackp;
}

constpush()
{
    Datum d;
    d.val = ((Symbol *)*pc++)->u.val;
    push(d);
}

varpush()
{
    Datum d;
    d.sym = (Symbol *)*pc++;
    push(d);
}

whilecode()

```

```

    pc = *((Inst **)(savepc+2)); /* next stmt */
}
bltin()
{
    Datum d;
    d = pop();
    d.val = *((double (*)(*))(*pc++))(d.val);
    push(d);
}

eval() /* Evaluate variable on stack */
{
    Datum d;
    d = pop();
    if (d.sym->type != VAR && d.sym->type != UNDEF)
        execerror("attempt to evaluate non-variable", d.sym->name);
    if (d.sym->type == UNDEF)
        execerror("undefined variable", d.sym->name);
    d.val = d.sym->u.val;
    push(d);
}

add()
{

```

```

{
    Datum d;
    Inst *savepc = pc; /* loop body */
    execute(savepc+2); /* condition */
    d = pop();
    while (d.val) {
        execute(*((Inst **)(savepc))); /* body */
        execute(savepc+2);
        d = pop();
    }
    pc = *((Inst **)(savepc+1)); /* next statement */
}

ifcode()
{
    Datum d;
    Inst *savepc = pc; /* then part */
    execute(savepc+3); /* condition */
    d = pop();
    if (d.val)
        execute(*((Inst **)(savepc)));
    else if (*((Inst **)(savepc+1))) /* else part? */
        execute(*((Inst **)(savepc+1)));
}

```

```

    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val += d2.val;
    push(d1);
}

sub()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val -= d2.val;
    push(d1);
}

mul()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val *= d2.val;
    push(d1);
}

```

```

div()
{
    Datum d1, d2;
    d2 = pop();
    if (d2.val == 0.0)
        execerror("division by zero", (char *)0);
    d1 = pop();
    d1.val /= d2.val;
    push(d1);
}

negate()
{
    Datum d;
    d = pop();
    d.val = -d.val;
    push(d);
}

gt()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val > d2.val);
}

```

```

    d1 = pop();
    d1.val = (double)(d1.val <= d2.val);
    push(d1);
}

eq()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val == d2.val);
    push(d1);
}

    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val != d2.val);
    push(d1);
}

and()
{

```

```

    push(d1);
}

lt()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val < d2.val);
    push(d1);
}

ge()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val >= d2.val);
    push(d1);
}

le()
{
    Datum d1, d2;
    d2 = pop();

```

```

    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val != 0.0 && d2.val != 0.0);
    push(d1);
}

or()
{
    Datum d1, d2;
    d2 = pop();
    d1 = pop();
    d1.val = (double)(d1.val != 0.0 || d2.val != 0.0);
    push(d1);
}

not()
{
    Datum d;
    d = pop();
    d.val = (double)(d.val == 0.0);
    push(d);
}

power()

```

```

{
    Datum d1, d2;
    extern double Pow();
    d2 = pop();
    d1 = pop();
    d1.val = Pow(d1.val, d2.val);
    push(d1);
}

assign()
{
    Datum d1, d2;
    d1 = pop();
    d2 = pop();
    if (d1.sym->type != VAR && d1.sym->type != UNDEF)
        execerror("assignment to non-variable",
            d1.sym->name);
    d1.sym->u.val = d2.val;
    d1.sym->type = VAR;
    push(d2);
}

print()
{
    Datum d;

```

```

    d = pop();
    printf("\t%.8g\n", d.val);
}

prexpr() /* print numeric value */
{
    Datum d;
    d = pop();
    printf("%.8g\n", d.val);
}

Inst *code(f) /* install one instruction or operand */
Inst f;
{
    Inst *oprogp = progp;
    if (progp >= &prog[NPROG])
        execerror("expression too complicated", (char *)0);
    *progp++ = f;
    return oprog;
}

execute(p)
Inst *p;
{
    for (pc = p; *pc != STOP; )

```

```

#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double Log(), Log10(), Sqrt(), Exp(), integer();

static struct { /* Keywords */
    char *name;
    int kval;
} keywords[] = {
    "if", IF,
    "else", ELSE,
    "while", WHILE,
    "print", PRINT,
    0, 0,
};

static struct { /* Constants */
    char *name;
    double cval;
} consts[] = {
    "PI", 3.14159265358979323846,
    "E", 2.71828182845904523536,
    "GAMMA", 0.57721566490153286060, /* Euler */
    "DEG", 57.29577951308232087680, /* deg/radian */

```

```

"PHI", 1.61803398874989484820, /* golden ratio */
0, 0
};

static struct {
    char *name;
    double (*func)();
} builtins[] = {
    "sin", sin,
    "cos", cos,
    "atan", atan,
    "log", Log, /* checks argument */
    "log10", Log10, /* checks argument */
    "exp", exp,
    "sqrt", Sqrt, /* checks argument */
    "int", integer,
    "abs", fabs,
    0, 0
};

init() /* install constants and built-ins in table */
{
    int i;
    Symbol *s;
    for (i = 0; keywords[i].name; i++)

        install(keywords[i].name, keywords[i].kval, 0.0);
    for (i = 0; consts[i].name; i++)
        install(consts[i].name, VAR, consts[i].cval);
    for (i = 0; builtins[i].name; i++) {
        s = install(builtins[i].name, BLTIN, 0.0);
        s->u.ptr = builtins[i].func;
    }
}

```
