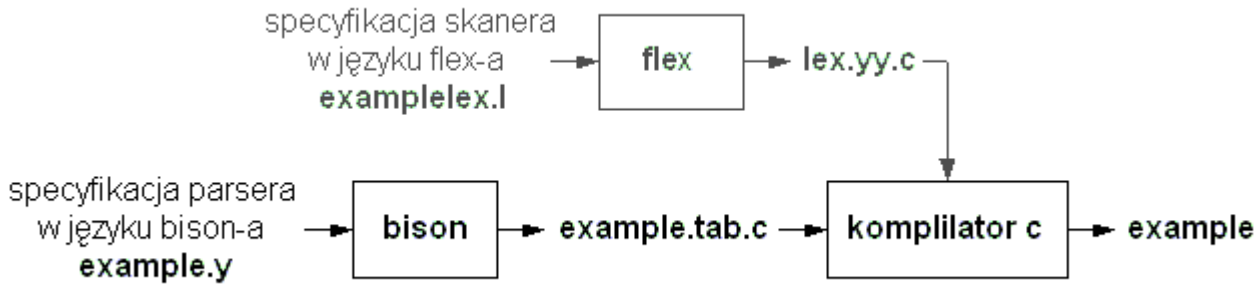
Bison - generator analizatorów składniowych

Spis treści:

Wprowadzenie

Bison jest generatorem parserów typu LALR. Bison generuje funkcję **analizatora składniowego**, która współpracuje z dostarczoną przez użytkownika funkcją **analizatora leksykalnego**. Skaner znajduje w strumieniu wejściowym podstawowe jednostki leksykalne zwane **tokenami** i na żądanie dostarcza je parserowi. Parser układa kolejne tokeny w struktury opisane **regułami gramatycznymi**. Gdy uda się znaleźć skrukturę odpowiadającą jakiejś regule, wykonywany jest kod dołączony do tej reguły, zwany **akcją**.

Przy pomocy bison-a można tworzyć parsery w sposób pokazany na poniższym rysunku. Wygenerowany przez bison-a parser wymaga podania kodu skanera. Najczęściej kod skanera jest generowany przy pomocy flex-a.



Specyfikacja

Specyfikacja dla bison-a ma układ podobny do specykicji flex-a:

```
{deklaracje}
%%
{reguły}
%%
{procedury}
```

Deklaracje

W sekcji deklaracje pomiędzy znakami %{ i %} umieszczany jest kod C, który ma być

wstawiony bezpośrednio na początku generowanego programu. Następnie wpisywane są deklaracje **tokenów**. Nazwy tokenów zwyczajowo pisane sa duzymi literami.

```
%token NAZWA1 NAZWA2 NAZWA3
```

Reguły

To najważniejsza sekcja. Tu umieszczamy reguły gramatyczne.

Ogólna postać reguły jest następująca:

```
<lewa strona>  : <alternatywa 1>   {akcja 1}
               | <alternatywa 2>   {akcja 2}
               ...
               | <alternatywa n>   {akcja n}
               ;
```

Przykładowo produkcje, które zapisywane na ćwiczeniach jako:

```
E --> E + T | T
T --> 0 | ... | 9
```

w bison-ie można zapisać jako:

```
wyr  :  wyr '+' term
     |  term
     ;
term : CYFRA
     ;
```

Symbole terminalne pojawiające się w naszej gramatyce są dwojakiego rodzaju:

1. zadeklarowane przy pomocy (`%token NAZWA_TERMINALA`);
2. pojedynczy znak w apostrofach (literał), np. '+'.

Nazwa niezadeklarowana jako token traktowna jest jako **nieterminal**. Każdy z symboli nieterminalnych musi przynajmniej raz pojawić się po lewej stronie reguły.

Z każdą regułą skojarzona może być **akcja semantyczna** (kod w C obowiązkowo zawarty w nawiasach `{` i `}`). Akcja jest wykonywana w momencie, gdy parser napotka strukturę, która została określona przez regułę. Akcje można umieszczać także wewnątrz reguł.

Akcje mogą zwracać wartości i korzystać z wartości zwróconych przez inne akcje. Do tego celu wykorzystuje się zmienne `$$`, `$1`, `$2`, `$3`... Zmienna `$$` przypisuje wartość do symbolu po lewej stronie reguły, zaś zmienne `$1`, `$2`, `$3`... zawierają wartości odpowiadające kolejnym składnikom reguły, np.

```
wyr : wyr '+' term { $$ = $1 + $3; }
/*
  $$ oznacza wartość zwracaną przez akcję,
  $1 oznacza wartość skojarzoną z symbolem wyr (po prawej stronie produkcji),
  $3 oznacza wartość skojarzoną z symbolem term,

  podstawienie $$ = $1 + $3 spowoduje, że akcja zwróci wartość będącą sumą
  wartości skojarzonych ze składnikami 1 i 3
*/
```

Jeżeli nie zdefiniowano inaczej domyślnie wartością reguły staje się wartość

pierwszego składnika, tj.

```
A : B { $$ = $1; } ;     oznacza to samo co     A : B ;
```

Procedury pomocnicze

Użytkownik musi dostarczyć bison-owi analizator leksykalny odpowiedzialny za bezpośrednie kontrolowanie wejścia i dostarczanie symboli terminalnych do parsera. Taką funkcję można wygenerować przy pomocy flex-a lub napisać własnoręcznie (`int yylex()`). Funkcja skanera zwraca na żądanie parsera numer kolejnego tokenu. Konkretna wartość danego tokenu jest dostarczana parserowi jako atrybut poprzez predefiniowaną zmienną `yylval`.

Wewnątrz sekcji procedury pomocnicze często umieszczane są procedury obsługi błędów.

Komantarze C (np. `/* komentarz */`) mogą być umieszczone w dowolnym miejscu specyfikacji.

Prosty przykład

Poniżej znajduje się przykład specyfikacji bardzo prostego parsera.

```
/* example.y - bardzo prosty parser */

/* wszystkie nazwy tokenow majace wiecej niz jeden znak musza byc zadeklarowane */
%token LETTER

%%

/* gdy na wejsciu pojawi sie symbol terminalny NUM zostanie wywolana ponizsza akcja */

input: LETTER { printf("rozpoznano -%c\n", $1); };

/* do przekazywania wartosci semantycznych pomiedzy regulami sluza tzw. pseudozmienne;
   akcja moze zwrocic wartosc przez podstawienia pseudozmiennej $$;
   pseudozmiennej $1 uzyto w przykladzie, aby wypisac wartosc tokenu NUM */

%%
#include "lex.yy.c"

main() {

  /* aby rozpoczac parsing trzeba wywolac funkcje yyparse */
  yyparse();
}

/* gdy funkcja yyparse wykryje blad wywoluje funkcje yyerror */
int yyerror(char *s) {

   printf("blad: %s\n", s);
}
```

Źródła: [example.y]

```
/* examplelex.l - skaner wspolpracujacy z parserem example.y */

/* zmienna globalna yylval sluzy do przekazywania parserowi atrybutu tokenu
   (wartosci semantycznej); zmienna yylval powinna jest typu YYSTYPE */
%%
-[abc]   {
           yylval=yytext[1];
           return LETTER;
         }
```

```
/* po dopasowaniu ciagu wejsciowego do yylval podstawiamy znak znajdujacy sie po -,
   funkcja yylex konczy swoje dzialanie i zwraca numer dopasowanego tokenu */

.|\n      ;
```

Źródła: examplelex.l

Uruchamianie

Przygotowany plik `example.y` podajemy na wejście bisona. W wyniku otrzymujemy plik `example.tab.c` zawierający kod parsera. Opcja `-d` powoduje wygenerowanie pliku nagłówkowego `example.tab.h` zawierającego deklaracje tokenów (dla celów komunikacji parsera ze skanerem).

```
bison -d example.y
```

Przygotowujemy plik `examplelex.l` i generujemy kod skanera `lex.yy.c` poleceniem:

```
flex examplelex.l
```

Kompilujemy wygenerowany wcześniej kod poleceniem:

```
gcc -o example example.tab.c -lfl
```

Otrzymujemy program wynikowy `example`. Możemy go uruchomić przykładowo podając na wejście `plik_testowy`:

```
./example <plik_testowy
```

Rozstrzyganie niejednoznaczności

Konflikty shift/reduce i reduce/reduce

Poniżej pokazano przykład niejednoznacznej gramatyki:

```
%token NUM
%%
expr : expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '(' expr ')'
| NUM
;
```

W podanej gramatyce wyrażenie:

```
exp - exp - exp
```

można wyprowadzić na dwa sposoby:

```
(exp - exp) - exp    -> wiązanie lewostronne
 exp - (exp - exp)   -> wiązanie prawostronne
```

W przypadku niejednoznaczności gramatyki podczas parsingu mogą zaistnieć dwie szczególne sytuacje:

- konfilkt **shift/reduce** - gdy parser staje przed wyborem dwóch poprawnych akcji: przesunięcia i redukcji,

- konflikt **reduce/reduce** - gdy parser ma do wyboru dwie poprawne akcje redukcji.

W obu przypadkach `bison` wygeneruje poprawny parser w oparciu o następujące zasady:

- konflikt shift/reduce rozstrzygany jest na korzyść operacji **shift** (wprowadzenie nowego symbolu ma pierwszeństwo w stosunku do zaakceptowania reguły),
- konflikt reduce/reduce rozstrzygany jest na korzyść **reguły gramatycznej występującej wcześniej** w specyfikacji

`Bison` zawsze informuje o liczbie konfliktów, które zostały automatycznie rozstrzygnięte. Zaleca się unikania konfliktów reduce/reduce.

Rozważmy inny przykład niejednoznacznej gramtyki.

```
instrukcja : IF_ wyrazenie instrukcja                  /* produkcja 1 */
           | IF_ wyrazenie instrukcja ELSE_ instrukcja  /* produkcja 2 */
           ;
```

Po wczytaniu `IF_ wyrazenie instrukcja` parser napotyka na konflikt *shift/reduce*. Ma do wyboru, albo zredukować wg produkcji 1 albo wczytać `ELSE_`. Ponieważ wczytanie nowego symbolu ma pierwszeństwo w stosunku do redukcji parser wczyta `ELSE_ instukcja`, a następnie zastosuje redukcję wg produkcji 2.

Rodzaje wiązań i priorytety

Kolejną sytuacją, która często prowadzi do konfliktów jest użycie operatorów unarnych. Ogólna postać reguły dla operatorów unarnych jest następująca:

```
exp : OP_UNARNY exp;.
```

W wielu przypadkach operatory unarne mają wyższy priorytet niż operatory binarne i ta zasada powinna znaleźć swoje odbicie w gramatyce lub podczas parsingu. Rozwiązaniem tego typu niejednoznaczności, które udostępnia bison, jest możliwość już na wstępie wyspecyfikowania priorytetu i sposobu wiązania operatorów. Priorytet i sposób wiązania dla symboli terminalnych jest określany w sekcji deklaracji. Do wyznaczenia sposobu wiązania wykorzystuje się słowa kluczowe:

```
%left - oznacza wiązanie lewostronne,
%right - oznacza wiązanie prawostronne,
%nonassoc - oznacza, że operator nie wiąże się ani lewostronnie,
            ani prawostronnie.
```

Przykład:

```
%nonassoc '<' '>' '=' GEQ LEQ NEQ
/* relational operators */

%left '+' '-' OR
/* addition operators */

%left '*' '/' AND
/* multiplication operators */

%right NOT UMINUS
/* unary operators */
```

W powyższym zapisie wszystkie symbole umieszczone w tej samej linii mają ten sam priorytet. Kolejność linii odpowiada wzrostowi priorytetu.

```
%left '+' '-'     /* niższy priorytet */
%left '*' '/'     /* wyższy priorytet */
```

Priorytet można przypisać nie tylko do terminala, ale też do reguły. Każda reguła gramatyki ma domyślnie priorytet pierwszego terminala znajdujący się po jej prawej stronie. Priorytet ten może być zmieniony przy pomocy znacznika `%prec`. Przy rozwiązywaniu konfliktu *shift/reduce* `bison` wybiera akcję do wykonania w następujący sposób:

- jeżeli *symbol* ma wyższy priorytet niż *reguła* wybierana jest operacja *shift*,
- jeżeli *reguła* ma wyższy priorytet niż *symbol* wybierana jest operacja *reduce*,
- jeżeli symbol i reguła mają taki sam priorytet typ wiązania symbolu determinuje rodzaj wykonywanej akcji;
    - jeżeli symbol wiąże się *lewostronne* wykonywana jest operacja *reduce*,
    - jeżeli symbol wiąże się *prawostronne* wykonywana jest operacja *shift*,
    - jeżeli dla symbolu *wiązanie jest niedozwolone* zgłaszany jest *błąd*.

Domyślnie `bison` preferuje lewostronną rekursję w regułach gramatycznych.

Zmianę priorytetu przy pomocy znacznika `%prec` wykonuje się umieszczając na końcu wybranej reguły zapis `%prec SYMBOL`, który przypisuje całej regule priorytet tokenu `SYMBOL`. Przykładowo, aby nadać operatorowi unarnego minusa najwyższy priorytet można napisać:

```
%token NUM
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
expr : expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '-' expr %prec UMINUS
| '(' expr ')'
| NUM
;
```

Token `UMINUS`, nie pojawia się nigdzie na wejściu, jest wykorzystywany jedynie do nadania odpowiedniego priorytetu unaranemu minusowi. Jeżeli nie skorzystalibyśmy ze znacznika `%prec` w powyższej regule, oba minusy (unarny i binarny) miałyby taki sam priorytet, ponieważ są reprezentowane przez ten sam znak z wejścia `('-')`.

Obsługa błędów

Obsługa błędów składniowych to ważny obszar związany z konstrukcją parserów. Zwykle nie chcemy, aby parser kończył pracę po napotkaniu pierwszego błędu. Zamiast tego parser po pojawieniu się błędu powinien próbować odzyskać kontrolę i znaleźć miejsce, od którego mógłby kontynuować pracę. Pyacc dostarcza pewnych mechanizmów do tzw. **error recovery**. Można wykorzystać specjalny predefiniowany **token error** do wskazania miejsca, gdzie mogą pojawić się błędy składniowe.

Normalnie w momencie, gdy parser znajduje na wejściu błędny symbol wypisuje komunikat o błędzie i rozpoczyna procedurę error recovery poprzez zdejmowanie ze stosu symboli dopóki nie natrafi na stan, który odpowiada akcji shift dla tokenu `error`. Jeżeli nie ma takiego stanu parser kończy pracę i zwraca wartość 1. Jeżeli znajdzie się taki stan, to wykonuje akcję shift na tokenie `error` i podejmuje na nowo parsing w specjalnym trybie obsługi błędu (error mode). W tym trybie pomija symbole, aż do momentu, gdy może wykonać legalną akcję shift. Aby pominąć lawinę komunikatów o błędach parser wraca do normalnego trybu po wykonaniu trzech legalnych operacji

shift.

Dla przykładu rozważmy regułę:

```
        instr :  error ';'   { yyerrok; }
```

Załóżmy, że pojawia się błąd składniowy podczas, gdy parsowany jest nieterminal `instr`. Parser wypisuje komunikat o błędzie i zdejmuje symbole ze stosu dopóki nie natrafi na token `error`. Wtedy woła procedurę `yyerrok` powodującą kontynuację parsingu w normalnym trybie. Opisane wyżej podejście do odzyskiwania kontroli po napotkaniu błedu, tzw. tryb paniki, sprawdza się dobrze w schemacie, gdzie wszystkie instrukcje są zakończone średnikami.

Przykłady

Przykład 1

Poniżej umieszczonony jest przykład specyfikacji bison-a wykorzystujący fragment gramatyki wyrażeń arytmetycznych.

```
/* exp.y - wygenerowany parser bedzie wykonywal odejmowanie */

%{
#include <stdio.h>
%}

%token NUM

%%
input: /* empty */
     | input line
     ;

line: '\n'
    | exp '\n'    { printf("%d\n", $1); }
    ;

exp: NUM
    | exp '-' exp        { $$ = $1 - $3; }
    | '(' exp ')'       { $$ = $2; }
    ;

/* Jezeli na wejsciu parsera podamy ciag 4-3-1, to otrzymamy 2. Dlaczego?
   Co zrobic, aby parser wypisywal poprawny wynik, tutaj 0? */

/* Nawet jesli podczas analizy gramatyki bison wykryje konflikt, to mimo to wygeneruje p
   - konflikt shift/reduce jest rozstrzygany na korzysc shift
   - konflikt reduce/reduce jest rozstrzygany na korzysc reguly wystepujacej wczesniej
     w specyfikacji */


%%
#include "lex.yy.c"

int yyerror(char *s) {

   fprintf(stderr, "blad: %s\n", s);
}

main() {

  yyparse();
}
```

Źródła: exp.y

```
/* explex.l - skaner wspolpracujacy z parserem exp.y */

%%
[ \t]   ;
[0-9]+   yylval = atoi(yytext); return NUM;
.|\n    return *yytext;
```

Źródła: explex.l

Przykład 2

Poniżej pokazano przykład parsera, który działa jak prosty kalkulator. Czyta wyrażenia arytmetyczne, złożone ze stałych rzeczywistych, operatorów `+, -, *, /,`, minus unarnego oraz nawiasów, ze standardowego wejścia i wypisuje wyniki na standardowe wyjście. Pojedyncze litery oznaczają zmienne (nie są rozróżniane małe i duże litery). Zmienne mogą mieć przypisywane wartości w następujący sposób `<zmienna> = <wyrażenie>`. Przykład pochodzi z http://epaperpress.com/lexandyacc/

```
/* calc.y - prosty kalkulator */

%{
    #include <stdio.h>
    void yyerror(char *);
    int yylex(void);

    int sym[26];
%}

%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'

%%

program:
        program statement '\n'
        | /* NULL */
        ;

statement:
        expression                      { printf("%d\n", $1); }
        | VARIABLE '=' expression       { sym[$1] = $3; }
        ;

expression:
        INTEGER
        | VARIABLE                      { $$ = sym[$1]; }
        | expression '+' expression     { $$ = $1 + $3; }
        | expression '-' expression     { $$ = $1 - $3; }
        | expression '*' expression     { $$ = $1 * $3; }
        | expression '/' expression     { $$ = $1 / $3; }
        | '(' expression ')'            { $$ = $2; }
        ;

%%

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

int main(void) {
    yyparse();
}
```

Źródła: calc.y

```
/* calclex.l - skaner wspolpracujacy z parserem calc.y */
```

```
%{
    #include "calc.tab.h"
    #include <stdlib.h>
    void yyerror(char *);
%}

%%

[a-z]       {
                yylval = *yytext - 'a';
                return VARIABLE;
                }

[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

[-+()=/*\n]     { return *yytext; }

[ \t]   ;       /* skip whitespace */

.               yyerror("Unknown character");

%%

int yywrap(void) {
    return 1;
}
```

Źródła: calclex.l

Przykład 3

Poniższy przykład przedstawia jak zmienić domyślny typ tokenu z wykorzystaniem makra YYSTYPE.

```
/* char.y - przyklad ilustrujacy przedefiniowanie typu tokenu z int na char */

%{
#define YYSTYPE char
%}

%token T_CHAR
%%
input    :
         | input '\n'             { YYACCEPT; }
         | input znaki '\n'
         ;

znaki    :  znaki T_CHAR  { printf("Znaleziono: %c\n", $2); }
         |  T_CHAR         { printf("Znaleziono: %c\n", $1); }
         ;

%%
#include "lex.yy.c"

main(){
  yyparse();
}

int yyerror(char *s) {
   printf("blad: %s\n", s);
}
```

Źródła: char.y

```
/* charlex.l - skaner wspolpracujacy z parserem char.y */

%%
[a-zA-Z] yylval = yytext[0];  return(T_CHAR);
.|\n             return(yytext[0]);
```

Źródła: charlex.l

Inne materiały

Więcej informacji można znaleźć w:

- manualu bisona
- Lex and YACC primer/HOWTO
- "Yacc: Yet Another Compiler-Compiler", S.C. Johnson