Bison - obsługa błędów

Konflikty - przykład

Niejednoznacznie skonstuowana gramatyka parsera, generuje konflikty, a te z kolei często prowadzą do błędów działania parsera. Zobaczmy to dla przykładowego kalkulatora poniżej.

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

%union{
        float fval;
}

%token <fval> FLOAT
%token SIN COS TAN LOG LN

%type <fval> expression

%left '+' '-'
%left '*' '/'
%right UMINUS

%%

program:        program statement '\n'
                | program '\n'
                | /* pusta regula */
                ;

statement:      expression       { printf("wynik: %.4f\n", $1); }
                ;

expression:     FLOAT
                | expression '+' expression     { $$ = $1 + $3; }
                | expression '-' expression     { $$ = $1 - $3; }
                | expression '*' expression     { $$ = $1 * $3; }
                | expression '/' expression     { if($3 == 0){
                                                      yyerror("Nie można dzielić przez 0!");
                                                      $$ = 0;
                                                  } else {
                                                      $$ = $1 / $3;};
                                                }
                | '-' expression %prec UMINUS   { $$ = -$2; }
                | '(' expression ')'            { $$ = $2; }
                | SIN expression                { $$ = sin($2); }
                | COS expression                { $$ = cos($2); }
                | TAN expression                { $$ = tan($2); }
                | LOG expression                { $$ = log10($2); }
                | LN expression                 { $$ = log($2); }
                                ;
%%

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

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

Źródła: calc_konflikty.y

```
%{
    #include <stdlib.h>
    #include "calc.tab.h"
%}

%%

([0-9]+)|([0-9]+\.[0-9]+)|([0-9]+\.[0-9]*(e|E)[0-9]+)  {  yylval.fval = atof(yytext);
                                                           return FLOAT;
                                                        }
"sin"|"SIN"     { return SIN; }
"cos"|"COS"     { return COS; }
"tan"|"TAN"     { return TAN; }
"LOG"|"log"     { return LOG; }
"LN"|"ln"       { return LN; }
[-+()=/*\n]     { return *yytext; }
[ \t]           ; /* pomin */
.               ; /* pomin */

%%

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

Źródła: calc_konfliktylex.l

Błędny wynik działania programu:

```
sin 3
wynik: 0.1411
sin 2
wynik: 0.9093
sin 2 + sin 3
wynik: 0.8417
```

Co trzeba poprawić, aby uzyskać prawidłowy wynik?

```
sin 2 + sin 3
wynik: 1.0504
```

Gramatyka instrukcji warunkowej

Omówienie tematu konfliktów na przykładzie gramatyki dla instrukcji warunkowej można znaleźć w Gramatyki-instrukcji-warunkowych.pdf.

Obsługa błędów

Obsługa błędów składniowych to kolejny 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 kontynuować pracę. Bison dostarcza pewnych mechanizmów do tzw. **error recovery**.

W tym celu można skorzystać z predefiniowanego **tokenu error** do wskazania miejsca, gdzie mogą pojawić się błędy składniowe. Dla przykładu rozważmy instrukcję zakończoną średnikiem.

```
wejscie : wejscie instrukcja ';'
        | wejscie error ';'
        | wejscie ';'
        | /*pusta reguła */
        ;

instrukcja : TYP ID         /* deklaracja zmiennej */
           ;
```

Załóżmy, że pojawia się błąd składniowy podczas, gdy parsowany jest nieterminal `instrukcja`. Co dzieje się dalej?

W uproszczeniu można powiedzieć, że parser wypisuje komunikat o błędzie i zdejmuje symbole ze stosu dopóki nie natrafi na token `error`. Wtedy woła procedurę `yyerrok` powodującą przełaczenie na kontynuację parsingu w normalnym trybie.

W szczególności, gdy parser znajduje na wejściu błędny symbol wypisuje komunikat o błędzie i rozpoczyna procedurę wychodzenia z błędu 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 nie generować nadmiernej liczby komunikatów o błędach parser wraca do normalnego trybu po wykonaniu trzech standardowych operacji shift. Aby odzyskać kontrolę, przy mniejszej liczbie operacji shift niż domyślnie, możemy zastosować funkcję `yyerror(char *)`.

Opisane wyżej podejście do odzyskiwania kontroli po napotkaniu błędu nazywa się trybem paniki.

Dokumentacja bisona na temat error recovery.

Obsługa zmiennych

Poniżej pokazano rozszerzenie kalkulatora o obsługę zmiennych, z wykorzystaniem tablicy symboli. Pojedyncze litery oznaczają zmienne. Zmienne mogą mieć przypisywane wartości w następujący sposób:
`<zmienna> = <wyrażenie>`

```
/* calc_4.y - kalkulator z obsluga zmiennych */

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

    int sym[26];
%}

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


%%

program:
        program statement '\n'
        | /* pusta regula */
        ;

statement:
        expression                      { printf("%d\n", $1); }
        | VARIABLE '=' expression       { sym[$1] = $3; printf("%d\n", $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_4.y

```
/* calc_4lex.l - skaner wspolpracujacy z parserem calc_4.y */

%{
    #include "calc_4.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]   ;       /* pomin */

.               yyerror("Nieznany znak");

%%

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

Źródła: calc_4lex.l

Ćwiczenia

Ćwiczenie 1

Proszę przetestować kalkulator przykładu powyżej, z rozdziału Obsługa zmiennych, po kątem:

- błędów leksykalnych - np. niepoprawnych znaków na wejściu, typu #$%
- błędów składniowych - np. niepoprawnej składni, np. 4 5 +, (3 ln)
- błędów semantycznych- np. zakresów argumentów funkcji, np. 5 / 0, ln -10

Pliki źródłowe znajdują się na serwerze artemis:
https://artemis.wszib.edu.pl/~mmajew/przyklady/bison3/.

Źródła: calc_4.y, calc_4lex.l

Ćwiczenie 2

Proszę napisać parser, który rozpoznaje wąski podzbiór HTML-a, tj. znaczniki typu <HTML>, <BODY>, <B> i
. Dla poprawnego wejścia parser powinien wypisywać komunikat, że znaleziony został poprawny kod HTML. Dla przykładowego poprawnego pliku:

```
<html>
<body>
Grammars for yacc are described using a variant of <b>Backus Naur Form</b> (BNF).<br>
This technique was pioneered by John Backus and Peter Naur.<br>
A BNF grammar can be used to express <b>context-free languages</b>.<br>
Most constructs in modern programming languages can be represented in <b>BNF</b>.
</body>
</html>
```

Źródła: test_html_1

na wyjściu powinien pojawić się komunikat: "Plik jest poprawny".

Ćwiczenie 3

Proszę rozszerzyć parser z poprzedniego zadania, tak aby obsłużył błąd składniowy dla kodu HTML pojawiającego się na wejściu. Np. dla niepoprawnego wejścia:

```
<html>
<body>
Grammars for yacc are described using a variant of Backus Naur Form</b> (BNF).<br>
This technique was pioneered by John Backus and Peter Naur.<br>
A BNF grammar can be used to express context-free languages</b>.<br>
Most constructs in modern programming languages can be represented in BNF.</b>
</html>
```

Źródła: test_html_2

na wyjściu powinny pojawić się komunikaty o błędach:

```
Blad w linii 3: Niesparowany znacznik </b>
Blad w linii 5: Niesparowany znacznik </b>
Blad w linii 6: Niesparowany znacznik </b>
```

Ćwiczenie 4

Proszę stworzyć własny prosty walidator HTML. Proszę wziąć pod uwagę następujące znaczniki:

- bold, break **,
**
- paragraf ** **
- nagłówki **<h1></h1>,<h2></h2>**
- link ****

Dla przykładowego poprawnego pliku wejściowego:

```
<!DOCTYPE html>
<html>
<body>

<h1>HTML basics</h1>

<p>This is a <br> paragraph with a line break.</p>
<p><b>This text is bold.</b></p>

<h2>HTML Links</h2>

<p>HTML links are defined with the a tag:</p>
<a href="https://www.w3schools.com">This is a link to w3schools website</a>

</body>
</html>
```

Źródła: test_walidatora.html

na wyjściu powinien pojawić się komunikat, że plik jest poprawny.