Cuprins

1. QUEX, FLEX ȘI BISON 1

1.1. QUEX 1
1.2. FLEX 1
1.3. BISON 2
1.4. INSTALARE 2
1.4.1. Instalare Flex 2
1.4.2. Instalare Bison 3
1.5. UTILIZARE 3
1.5.1. Utilizare Flex 3
1.5.2. Utilizare Bison 4
1.6. GENERARE COD MIPS 5
1.7. EXECUTARE 7

2. BIBLIOGRAFIE 8

Documentaţie Compilator

Echipa Quex/Bison

January 14, 2021

1. Quex, Flex și Bison

1.1. Quex

Quex este un lexer/tokenizer considerat urmașul lui Flex, care generează cod C++ rapid și eficient. Într-un compilator, el este folosit pentru a citi cod dintr-un fișier text sau de la tastatură, cod care este mai apoi împărțit în tokeni care vor fi trimiși mai departe parserului (în cazul nostru Bison).

Câteva caracteristici ale Quex:

- Generează analizatoare lexicale codate direct, mai degrabă decât programe care se bazează pe tabele.
- Generează grafuri de tranziție de stare ale motoarelor generate.
- Are moduri care pot fi moștenite și ”mode transitions” care pot fi controlate.
- Are suport pentru analiză pe baza indentării (cu ajutorul tokenilor INDENT, DEDENT, NODENT).
- Are două strategii de pasare a tokenilor: ”queue” si ”single token”.

Împreună cu colegii mei de echipă, am încercat implementarea compilatorului nostru folosind Quex, însă datorită nenumăratelor erori generate de acesta, am renunțat și am ales folosirea Flex.

1.2. Flex

Flex este un program folosit pentru generarea de scannere: programe care recunosc pattern-uri lexicale în text. Flex citește fișierele de intrare(sau inputul standard de la tastatură dacă niciun fișier nu îi este oferit) și generează tokeni.

Flex generează ca și output un fișier C, ”lex.yy.c”, care definește o funcție yylex.

Câteva caracteristici ale Flex:

- Un analizor lexical în Flex are de obicei complexitatea O(n), n fiind lungimea inputului, deoarece execută un număr constant de operații pentru fiecare simbol primit ca input.

- Scannerul generat de Flex nu este ”reentrant”, adică multiple apelări nu pot rula concurent pe un sistem cu un singur procesor. Acest lucru poate genera probleme majore pentru programele care folosesc scannerul generat din diferite thread-uri. Pentru a depăși această problemă, există opțiuni pe care Flex le oferă pentru a dobândi această proprietate. O descriere detaliată a acestor opțiuni poate fi găsită în manualul Flex.
- Flex poate genera doar cod C sau C++.

Un fișier Flex constă din trei secțiuni, separate de o linie care conține doar ”%%”.

Definiții
%%
Reguli
%%
Cod scris de user

1.3. Bison

Bison este un program care citește un limbaj independent de context, avertizează asupra oricăror ambiguități la parsare și generează un parser care citește o secvență de tokeni și decide dacă secvența respectivă este conformă cu sintaxa specificată de gramatica limbajului.

De mult ori Bison este folosit împreună cu Flex pentru tokenizarea datelor de intrare și generarea de tokeni pentru Bison.

Câteva caracteristici ale Bison:

- Generarea de contraexemple → rezolvarea de conflicte necesită de obicei ceva experiență de la user. Contraexemplele ajută la înțelegerea rapidă a unor conflicte și pot demonstra că problema este că gramatica este de fapt ambiguă.
- În mod normal, Bison generează un parser care nu este ”reentrant”. Pentru a dobândi această proprietate, trebuie folosită declararea %define api.pure . Mai multe detalii despre această proprietate pot fi găsite în manualul de Bison.
- Bison poate genera cod pentru C, C++ și Java.

1.4. Instalare

Înainte de a putea folosi Flex și Bison sunt necesare câteva instalări și configurații.

1.4.1. Instalare Flex

Pentru a putea executa comenzile specifice lexerului Flex, va trebui să îl instalăm și să adăugăm calea către executabilul de Flex în Path-ul din Environment Variables pentru a putea utiliza comanda flex în cmd de oriunde, fără să mai fie nevoie să navigăm in folderul unde Flex este instalat.

Link descărcare: https://sourceforge.net/projects/winflexbison/

După configurarea path-ului, pentru a putea folosi comanda flex în cmd, mai este nevoie să redenumim în folderul descărcat executabilul win_flex.exe în flex.exe. Altfel în loc de comanda flex, ar trebui să scriem mereu win_flex.

1.4.2. Instalare Bison

În fișierul descărcat de pe linkul propus în secțiunea de mai sus, putem găsi și un executabil cu numele win_bison.exe. Asemănător cu ceea ce am făcut la instalarea Flex, și pentru Bison trebuie să configurăm path-ul și apoi să redenumim executabilul win_bison.exe în bison.exe pentru a folosi din cmd comanda bison, și nu comanda win_bison.

1.5. Utilizare

Utilizarea celor două tool-uri se poate face separat sau împreună.

1.5.1. Utilizare Flex

Pentru a realiza tokenizarea simbolurilor noastre vom scrie comenzile specifice Flex într-un fișier cu extensia ***.l**

Sintaxa Flex din acest fișier arată astfel:

```
%{
    #include <stdlib.h>
    #include "node.h" /* Must be here before y.yab */
    #include "y.tab.h"

    int yylineo;
    int line_no = 1;
    void yyerror(char* );

%}

%%
\/\/(.*)          ;
[1-9][0-9]*       { yylval.value = atoi(yytext); return NUMBER; }
[0]               { yylval.value = atoi(yytext); return NUMBER; }
[a-z]             { yylval.name = *yytext; return IDENTIFIER; }
[-/=+*)(;}{><]     { return *yytext; }
"\n"          {line_no+=1;     }
[ \t]+ ;
"+="      { return TKN_PLUS_EQ;    }
"-="      { return TKN_MINUS_EQ;    }
"*="      { return TKN_MULTIPLY_EQ;    }
"/="      { return TKN_DIVISION_EQ;    }
"if"      { return TKN_IF;     }
"while"   { return TKN_WHILE;  }
"do"      { return TKN_DO;     }
"int"     { return TKN_INT;    }
"else"    { return TKN_ELSE;   }
"switch"  { return TKN_SWITCH;    }
"case"    { return TKN_CASE;   }
"default" {return TKN_DEFAULT;   }
"break"   { return TKN_BREAK;  }
"=="      { return TKN_EQ;     }
"!="      { return TKN_NOT_EQ;    }
"&&"      { return TKN_AND;    }
"||"      { return TKN_OR;     }
"for"     { return TKN_FOR;    }
"<="      { return TKN_LESS_EQ;    }
">="      { return TKN_GREATER_EQ;    }
%%

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

În prima parte, cea de definiții, vom include headerul stdlib.h pentru a putea folosi funcția atoi() în partea de reguli. De asemenea, includem și headerele ”node.h”, care se ocupă de generarea arborelui de simboluri și headerul ”y.tab.h” care este un header generat de executarea comenzii bison pe fisierul nostru cu extensia ***.y** .

În partea de reguli sunt declarate regulile după care se generează tokenii. De exemplu, la întâlnirea cuvântului ”if”, lexerul va transmite către parser că a întâlnit tokenul potrivit pentru if.

Se va executa tokenizarea pentru fiecare simbol, iar fiecare token împreună cu valoarea acestuia (dacă această valoare există) vor fi trimise parserului Bison.

1.5.2. Utilizare Bison

Bison va primi tokenul și valoarea trimisă de către Flex și va construi arborele sintactic pe baza acestora. De reținut este faptul că Bison primește tokenii, nu invers!

Pentru a realiza parsarea vom scrie comenzile specifice Bison într-un fișier cu extensia specifică analizei sintactice ***.y** cum ar fi:

```
expr:
    NUMBER              { $$ = CreateConstant($1); }
    | IDENTIFIER              { $$ = CreateId($1); }
    | IDENTIFIER '=' expr     { $$ = CreateOperator('=', 2, CreateId($1), $3); }
    | IDENTIFIER TKN_PLUS_EQ expr     { $$ = CreateOperator(TKN_PLUS_EQ, 2, CreateId($1), $3); }
    | IDENTIFIER TKN_MINUS_EQ expr     { $$ = CreateOperator(TKN_MINUS_EQ, 2, CreateId($1), $3); }
    | IDENTIFIER TKN_MULTIPLY_EQ expr     { $$ = CreateOperator(TKN_MULTIPLY_EQ, 2, CreateId($1), $3); }
    | IDENTIFIER TKN_DIVISION_EQ expr     { $$ = CreateOperator(TKN_DIVISION_EQ, 2, CreateId($1), $3); }
    | expr '+' expr     { $$ = CreateOperator('+', 2, $1, $3); }
    | expr '-' expr     { $$ = CreateOperator('-', 2, $1, $3); }
    | expr '*' expr     { $$ = CreateOperator('*', 2, $1, $3); }
    | expr '/' expr     { $$ = CreateOperator('/', 2, $1, $3); }
    | expr TKN_LESS_EQ expr     { $$ = CreateOperator(TKN_LESS_EQ, 2, $1, $3); }
    | expr TKN_GREATER_EQ expr     { $$ = CreateOperator(TKN_GREATER_EQ, 2, $1, $3); }
    | expr '<' expr     { $$ = CreateOperator('<', 2, $1, $3); }
    | expr '>' expr     { $$ = CreateOperator('>', 2, $1, $3); }
    | expr TKN_EQ  expr     { $$ = CreateOperator( TKN_EQ, 2, $1, $3); }
    | expr TKN_NOT_EQ expr     { $$ = CreateOperator(TKN_NOT_EQ, 2, $1, $3); }
    | expr TKN_AND expr     { $$ = CreateOperator(TKN_AND, 2, $1, $3); }
    | expr TKN_OR  expr     { $$ = CreateOperator( TKN_OR, 2, $1, $3); }
    | '(' expr ')'      { $$ = $2; }
    ;
```

Bison va executa codul dintre acolade după reducerea expresiei respective.

Cele 3 funcții care sunt apelate între paranteze sunt declarate la finalul fișierului cu extensia *.y și sunt funcții care se ocupă cu crearea nodurilor pentru arborele sintactic în funcție de tipul găsit : număr, identificator sau operator.

```
105 Node* CreateConstant(int value)
106 {
107     Node* newNode = (Node*) malloc(sizeof(Node));
108 
109     newNode->type = CONST;
110     newNode->constant.value = value;
111 
112     return newNode;
113 }
114 
115 Node* CreateId(char name)
116 {
117     Node* newNode = (Node*) malloc(sizeof(Node));
118 
119     newNode->type = ID;
120     newNode->id.name = name;
121 
122     return newNode;
123 }
124 
125 Node* CreateOperator(int type, int noOfOpr, ...)
126 {
127     Node* newNode = (Node*) malloc(sizeof(Node) + noOfOpr * sizeof(Node*));
128 
129     newNode->type = OPR;
130     newNode->opr.noOfOpr = noOfOpr;
131     newNode->opr.type = type;
132 
133     va_list operands;
134     va_start(operands, noOfOpr);
135 
136     for (int i = 0; i < noOfOpr; ++i)
137         newNode->opr.operands[i] = va_arg(operands, Node*);
138 
139     va_end(operands);
140 
141     return newNode;
142 }
143 
```

1.6. Generare cod MIPS

În fișierul main.c, pe lângă funcția main() propriu-zisă, mai avem declarate câteva funcții care ne vor ajuta la scrierea codului în fișierul de output.

```
31  int getReserveRegistry()
32  {
33      /* Go through all of our registries and find a empty one */
34      for (int i = 2; i < 32; ++i)
35          if (FALSE == declared[i] && FALSE == visited[i])
36              return i;
37  }
38 
```

```
50  int registerIndex(char name)
51  {
52      return name - 'a' + 2; // First 2 are reserved
53  }
54
55  int getFreeAccumulator()
56  {
57      return visited[accumulator1] == FALSE ? accumulator1
58           : visited[accumulator2] == FALSE ? accumulator2
59           : visited[accumulator3] == FALSE ? accumulator3
60           : -1;
61  }
62
63  void FreeAccumulators()
64  {
65      visited[accumulator1] = FALSE;
66      visited[accumulator2] = FALSE;
67      visited[accumulator3] = FALSE;
68  }
```

De asemenea, în main.c este declarată și funcția Generate(), funcția care se ocupă de fapt de afișarea codului în fișierul de ieșire.

În funcție de tipul nodului (CONST, ID sau OPR) instrucțiunea switch va fi apelată corespunzător.

Mai jos este o poză din cod cu o mică bucată din funcția Generate().

```
switch (node->opr.type)
{
    case '-': fprintf(yyout, "sub $%d, $%d, $%d\n",  returnAcumul, leftReg, rightReg);                    break;
    case '+': fprintf(yyout, "add $%d, $%d, $%d\n",  returnAcumul, leftReg, rightReg);                    break;
    case '/': fprintf(yyout, "div $%d, $%d, $%d\n",  returnAcumul, leftReg, rightReg);                    break;
    case '*': fprintf(yyout, "mult $%d, $%d, $%d\n", returnAcumul, leftReg, rightReg);                    break;
    case TKN_NOT_EQ: fprintf(yyout, "xor $%d, $%d, $%d\n",  returnAcumul, leftReg, rightReg);             break;
    case TKN_OR : fprintf(yyout, "or $%d, $%d, $%d\n",   returnAcumul, leftReg, rightReg);                break;
    case TKN_AND: fprintf(yyout, "and $%d, $%d, $%d\n",  returnAcumul, leftReg, rightReg);                break;
    case '<': fprintf(yyout, "slt $%d, $%d, $%d\n",  returnAcumul, leftReg, rightReg);                    break;
    case TKN_PLUS_EQ: fprintf(yyout, "add $%d, $%d $%d\n",   leftReg, leftReg, rightReg);                 break;
    case TKN_MINUS_EQ: fprintf(yyout, "sub $%d, $%d, $%d\n",  leftReg, leftReg, rightReg);                break;
    case TKN_MULTIPLY_EQ: fprintf(yyout, "mult $%d, $%d, $%d\n", leftReg, leftReg, rightReg);             break;
    case TKN_DIVISION_EQ: fprintf(yyout, "div $%d, $%d, $%d\n",  leftReg, leftReg, rightReg);             break;
    case '=': fprintf(yyout, "li $%d, $%d\n",        leftReg, rightReg);                                  break;
```

1.7. Executare

După instalarea, setarea și scrierea comenzilor prezentate în capitolele precedente vom folosi următoarele comenzi pentru a executa tokenizarea în Flex, iar mai apoi parsarea în Bison.

Mai întâi se va naviga până în folderul unde sunt salvate fișierele cu extensiile *.l și *.y (în cazul nostru fișierele se numesc lexer.l și parser.y), asigurându-ne ca în același folder se află și fișierul node.h împreună cu fișierul main.c.

Se vor executa pe rând comenzile :

```
1  bison -y -d parser.y
2  flex lexer.l
3  gcc -g lex.yy.c main.c y.tab.c
```

Ca rezultat al acestor comenzi o să fie un fișier **a.exe** pe care îl putem executa, astfel executându-se generatorul de tokeni împreună cu parserul creat.

Pentru simplificarea utilizării, am creat un fisier cu extensia ***.bat** care poate fi executat, astfel utilizatorul fiind scutit de scrierea comenzilor de mai sus pe rând în consolă.

2. Bibliografie

- Quex:

  http://quex.sourceforge.net/doc/html/main.html

- Flex :

  https://www.cs.virginia.edu/~cr4bd/flex-manual/
  https://github.com/westes/flex/

- Bison

  https://www.gnu.org/software/bison/manual/html_node/index.html

- MIPS

  http://vbrunell.github.io/docs/MIPS%20Programming%20Guide.pdf

Împărțire sarcini echipe:

Flex / Quex

- Cojocaru Vladimir 10LF391,
- Duma Vlad Mihai 10LF391,
- Andrei Laurențiu-Adrian 10LF391
- Călăvie Adrian Constantin 10LF291
- Arhip Florin 10LF291
- Baciu Radu-Bogdan 10LF291

Bison

- Cojocaru Rareș 10lf391,
- Aglițoiu Marius 10lf391
- Popa Vlad-Valentin 10LF294,
- Serea Filip 10LF294,
- Timuș Andrei Flavian 10LF294

Pentru generarea codului MIPS am lucrat împreună toate echipele.