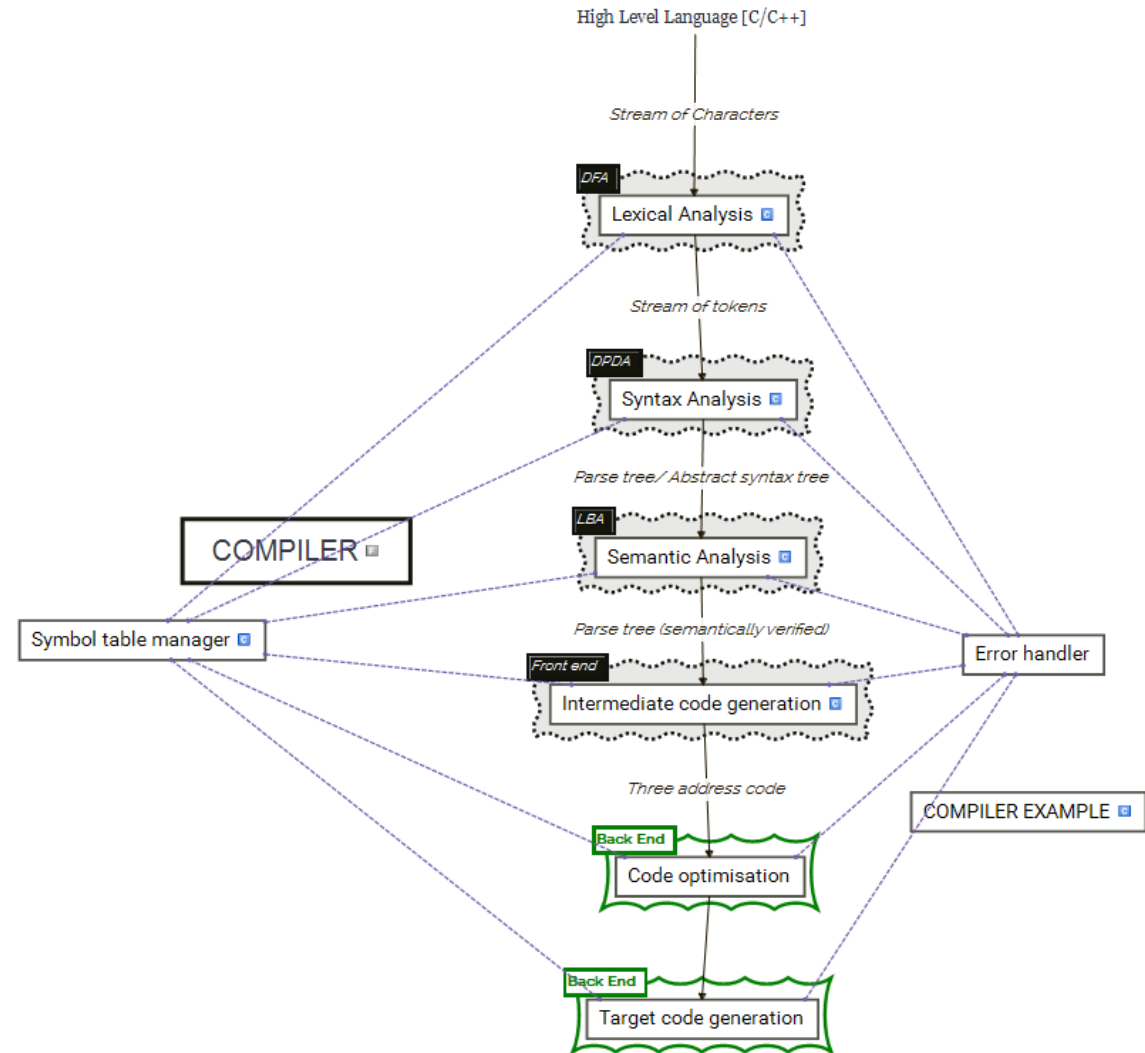
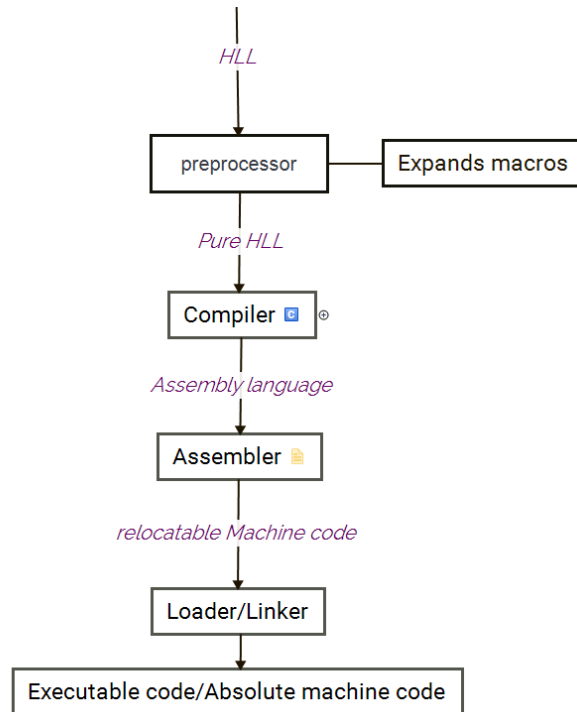


Yacc/Bison

Compiler overview



Introduction to Bison

- What is parser?
- Two basic parsers:
 - Top down
 - Bottom –up
- What are flex and bison?
- Why should you learn Flex/Bison pattern syntax when you could just write your own parser?
- What is the difference between lex/flex and yacc/bison?

Bison file structure

- A bison input file (bison grammar file) has the following structure with special punctuation symbols `%%`, `%{`, `%}`

```
%{  
Prologue e.g. C declaration  
%}  
bison declarations  
%%  
Grammar rules  
%%  
Epilogue e.g. Additional C code
```

Bison file structure -continued

C declaration ⊖ Preprocessing directives

`/* Infix notation calculator--calc */`

`%{`

`%{`

`#define YYSTYPE double`

`#include <math.h>`

`%}`

bison declarations ⊖ provides information to Bison about the token types

`/* BISON Declarations */`

`%token NUM`

`%left '-' '+'`

`%left '*' '/'`

`%left NEG /* negation--unary minus */`

`%right '^' /* exponentiation */`

⊖ Defining token codes

Establish associativity

setting up the global variables used to communicate between the scanner and parser

Bison file structure - continued

Grammar rules

```
/* Grammar follows */
%%
input:    /* empty string */
        | input line
        ;

line:     '\n'
        | exp '\n' { printf ("\t%.10g\n", $1); }
        ;

exp:      NUM          { $$ = $1;          }
        | exp '+' exp  { $$ = $1 + $3;    }
        | exp '-' exp  { $$ = $1 - $3;    }
        | exp '*' exp  { $$ = $1 * $3;    }
        | exp '/' exp  { $$ = $1 / $3;    }
        | '-' exp %prec NEG { $$ = -$2;   }
        | exp '^' exp  { $$ = pow ($1, $3); }
        | '(' exp ')'  { $$ = $2;         }
        ;
%%
```

Establish Operator precedence

Higher the line number, higher the precedence

Epilogue e.g. Additional C code

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

Flex File

- ```
/* Mini Calculator */
/* calc.lex */

%{
#include "heading.h"
#include "tok.h"
int yyerror(char *s);
int yylineno = 1;
%}

digit [0-9]
int_const {digit}+

%%

{int_const} { yylval.int_val = atoi(yytext); return INTEGER_LITERAL; }
"+" { yylval.op_val = new std::string(yytext); return PLUS; }
"*" { yylval.op_val = new std::string(yytext); return MULT; }

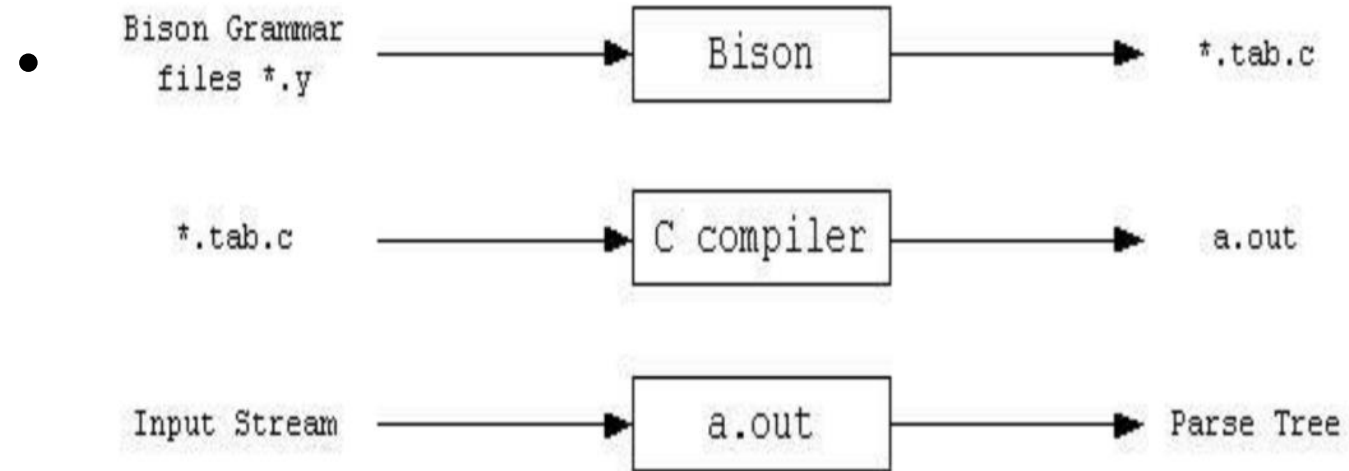
[\t]* {}
[\n] { yylineno++; }

. { std::cerr << "SCANNER "; yyerror(""); exit(1); }
```

Definitions ⊖ include declarations of constant, variable and regular definitions.

Rules ⊖ define the statement of form p1 {action1} p2 {action2}....pn {action}.

# How to execute bison file ?



.Compile the example.y file using bison.

command:  $\ominus$  bison -d example.y

example.tab.c.

output  $\ominus$

example.tab.h

Write the grammar specification for bison (example.y), including grammar rules, yyparse() and yyerror().



# Execution steps

- Write a lexical analyzer to process input and pass tokens to the parser

Run flex on the scanner specification to generate

command:  $\ominus$  flex exp.l

output  $\ominus$  lex.yy.c

- Compile the two .c files and link them together

example.tab.c.

input:

lex.yy.c

command:

$\ominus$  gcc lex.yy.c example.tab.c -ll -o xyz

output:

$\ominus$  xyz executable file

- Run

./xyz

A parse tree will be created