Automating Syntax analysis

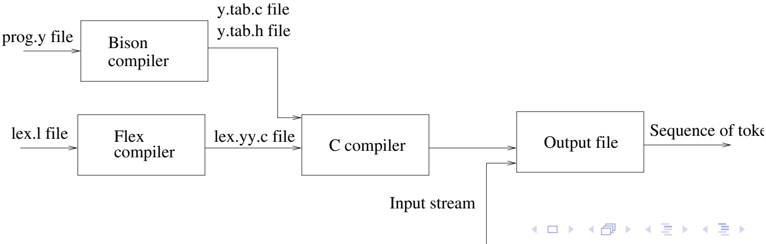
Sudakshina Dutta

Indian Institute of Technology Goa

23^{rd} February, 2022

Parser using Bison

- Flex and Bison help in writing programs that transform structured input
- As the input is divided into tokens by Flex, a program often needs to find relationship among the token
- It is done by grammar rules
- Bison takes the grammar and produces parsing routines written in C
- Bison uses shift-reduce parsing mechanism
- Recursion can be handled efficiently



prog.y file

Bison compiler

y.tab.c file

y.tab.h file

lex.l file

Flex compiler

lex.yy.c file

C compiler

Output file

Sequence of toke

Input stream

Structure of a Bison file

- The structure of a Bison parser is similar to that of a Flex lexer
- The first section is the declaration section which has a literal code block enclosed in “%{” and “%}”
- It is followed by definitions of all the tokens we expect to receive from lexical analyzer
- The first %% indicates the beginning of the rules section
- The second %% indicates the end of the rules and end of the user subroutine section

Structure of a Bison file

```
%{
C Declarations
%}
Bison Declarations
%%
Grammar Rules
%%
Additional C Code
```

- An example consisting of Flex and Bison programs

```
%{

#include < stdio.h >
#include" parser.tab.h"

}%


%%

          [\t] ;
          [A − Z0 − 9]* {return TEXT; };
           “begin” {return BEGIN; };
           “end” {return END; };
           \n {return yytext[0]; };
           . {return yytext[0]; };




%%
```

```
%{

#include < stdio.h >
int yylex(void);

}%
%token BEG END TEXT

%%

          paragraph : BEG TEXT END‘\n′{printf(" The paragraph is valid\n"); } ;

%%

int main(){
          yyparse();
}
int yyerror(char * s)
{
          fprintf(stderr, “%s\n”, s);

}
```

- **Compilation steps**:

  ```
  bison -d prog.y
  flex prog.l
  cc prog.tab.c lex.yy.c -ly -lfl
  ```

- **Declaration Section :** It introduces any initial C program code we want copied into the final program
- The declaration section also includes names of the tokens which is used by the lexer and the parser

  Example : %*token NAME NUMBER*

Rules Section

- It describes the actual grammar as a set of production rules

  Example : $expression : NUMBER' +' NUMBER \mid NUMBER' -' NUMBER;$

- Each rule has single name on the left hand side of a ":" operator, a list of symbols and action code on the right hand side
- A semicolon indicates end of the rule
- The first rule is the highest level rule

  — The parser attempts to find a list of tokens which match the initial rule

User Subroutine Section

- The most important subroutine is *main*() which repeatedly calls *yyparse*() until lexer's input file runs out
- In case of syntax error, the parser calls *yyerror*(). Error recovery code can be provided which tries to get the parser back into a state from where it can continue parsing

Passing values from lexer to parser

- yyparse() returns a value of 0 if the input it parses is valid according to the given grammar rules. It returns a 1 if the input is incorrect and error recovery is impossible
- yyparse() does not do its own lexical analysis. It calls a routine called yylex() everytime it wants to obtain a token from the input
- yylex() returns a value indicating the type of token that has been obtained. If the token has an actual value, this value or some representation is returned in an external variable named yylval

References

- https://web.iitd.ac.in/s̃umeet/flex__bison.pdf
- https://www.gnu.org/software/bison/manual/bison.html#Language-and-Grammar