Understanding parser.y and Scanner-Parser Connection

Table of Contents

Overview

This document explains how the **Avian compiler's parser** works, focusing on: - The `parser.y` file structure and every keyword - How `scanner.l` connects to `parser.y` - How tokens flow from lexical analysis to syntax analysis - The generated files (`y.tab.h` and `y.tab.c`)

File Structure

```
avian-main/
   scanner.l          # Lexical analyzer (tokenizer)
   parser.y           # Syntax analyzer (parser)
   lex.yy.c          # Generated scanner C code (from scanner.l)
   y.tab.h           # Generated token definitions (from parser.y)
   y.tab.c           # Generated parser C code (from parser.y)
```

Parser.y Detailed Explanation

Section 1: Prologue (Lines 1-10) - C Declarations

```
%{
#include <stdio.h>
#include <stdlib.h>

extern int yylineno;
extern char* yytext;

int yylex(void);
```

```
void yyerror(const char *s);
%}
```

Line-by-Line Breakdown:

Line	Code	Purpose
1	`%{`	Opens C code prologue block - everything here is copied to generated C file
2-3	`#include`	Standard C libraries for I/O and memory management
5	`extern int yylineno;`	**Global variable from scanner** - tracks current line number for error reporting
6	`extern char* yytext;`	**Global variable from scanner** - points to current token text
8	`int yylex(void);`	**THE BRIDGE FUNCTION** - parser calls this to get next token from scanner
9	`void yyerror(const char *s);`	Error handler function - called when syntax error occurs
10	`%}`	Closes C code prologue block

Why `extern` is used:

- `extern` tells the compiler these variables are **defined in another file** (scanner)
- The scanner (`lex.yy.c`) provides the actual implementation
- This allows parser to access scanner's information

The Critical Connection: `yylex()`

```
int yylex(void);
```

This is THE BRIDGE between scanner and parser: - **Parser calls** `yylex()` whenever it needs the next token - **Scanner implements** `yylex()` in `lex.yy.c` (generated by Flex) - **Returns:** An integer token code (e.g., AVN_INT = 258) - **Side effects:** Updates `yytext` (token text) and `yylineno` (line number)

Flow:

```
Parser: "I need next token"
```

```
  ↓
Calls: yylex()
  ↓
Scanner: Reads input, matches pattern
  ↓
Returns: Token code (e.g., 258 for AVN_INT)
  ↓
Parser: "Got AVN_INT token, continue parsing"
```

Section 2: Token Declarations (Lines 12-42)

The `%token` Keyword

```
%token AVN_INT AVN_IF AVN_ELSE AVN_WHILE AVN_PRINT
```

Purpose: - Declares **terminal symbols** (tokens) that the parser recognizes - Each token gets a **unique integer code** (defined in `y.tab.h`) - Example: `AVN_INT = 258`, `AVN_IF = 259`, etc.

Connection to Scanner: When scanner matches a pattern, it **returns these token codes**:

```
// In scanner.l (line 44):
avn_int     { print_token("KEYWORD", yytext); return AVN_INT; }
//                                              ^^^^^^^^^^^^^^
//                                             Returns integer 258
```

All Token Categories:

```
/* Keywords - Language reserved words */
%token AVN_INT AVN_IF AVN_ELSE AVN_WHILE AVN_PRINT
%token AVN_FOR AVN_BREAK AVN_CONTINUE AVN_FUNC AVN_RETURN
%token AVN_CIN AVN_COUT AVN_SCAN AVN_FLOAT AVN_STR AVN_MAIN AVN_END

/* Identifiers and Literals */
%token ID NUM STRING        // Variables, numbers, strings

/* Assignment Operator */
%token ASSIGN               // =

/* Delimiters */
%token LPAREN RPAREN        // ( )
%token LBRACE RBRACE        // { }
%token SEMI COMMA           // ; ,

/* Avian Custom Operators */
%token OP_ADD OP_SUB        // _+_ _-_
```

```
%token OP_MUL OP_DIV        // _*_ _/_
%token OP_INC OP_DEC        // _++_ _--_

/* Avian Custom Punctuations */
%token PUNC_COMMA           // _,_
%token PUNC_SEMI_APOS       // ;'
%token PUNC_COLONPAIR       // ::

/* Comparison Operators */
%token LT_OP GT_OP          // < >
%token EQ_OP NE_OP          // == !=

/* Stream Operators */
%token EXE_OP INS_OP        // << >>
```

Operator Precedence & Associativity

```
%left OP_ADD OP_SUB
%left OP_MUL OP_DIV
```

What this means: - `%left`: Left-associative operators - `a + b + c` is parsed as `(a + b) + c` - **Order matters**: Lower line = **higher precedence** - `OP_MUL` and `OP_DIV` have higher precedence than `OP_ADD` and `OP_SUB` - So `2 + 3 * 4` correctly parses as `2 + (3 * 4)`

Alternative directives: - `%right` - Right associativity (e.g., for exponentiation: `2^3^4` = `2^(3^4)`) - `%nonassoc` - Non-associative (error if used twice: `a < b < c` would be invalid)

Start Symbol

```
%start P
```

- **Defines the root of the parse tree**
- Parsing begins here
- `P` stands for **Program** - the top-level grammar rule
- **Must eventually reduce to this symbol for successful parse**

Section 3: Grammar Rules (Lines 45-96)

```
%%
```

The %% delimiter: Marks the beginning of grammar rules section

Grammar Rule Structure

```
P   : B
      {
        printf("Syntax analysis successful\n");
      }
    ;
```

Components: - `P`: Non-terminal (left-hand side) - `:`: "Is defined as" / "Produces" - `B`: Production (right-hand side) - `{ ... }`: **Semantic action** - C code executed when rule matches - `;`: Ends the rule

Translation: "A Program (P) consists of a Block (B)"

Complete Grammar Hierarchy

```
/* Program → Block */
P   : B
      { printf("Syntax analysis successful\n"); }
    ;

/* Block → { StatementList } */
B   : LBRACE SL RBRACE
    ;

/* StatementList → Statement StatementList |   (empty) */
SL  : S SL
    | /* epsilon - empty production */
    ;

/* Statement → Declaration | Assignment | If | While | Output */
S   : D
    | A
    | IF
    | WH
    | OUT
    ;

/* Declaration → avn_int ID; | avn_int ID = Expression; */
D   : AVN_INT ID SEMI
    | AVN_INT ID ASSIGN E SEMI
    ;

/* Assignment → ID = Expression; */
A   : ID ASSIGN E SEMI
    ;

/* If Statement → avn_if (E) B | avn_if (E) B avn_else B */
IF  : AVN_IF LPAREN E RPAREN B
```

```
    | AVN_IF LPAREN E RPAREN B AVN_ELSE B
    ;

/* While Loop → avn_while (E) B */
WH  : AVN_WHILE LPAREN E RPAREN B
    ;

/* Output → avn_print ID; | avn_print Expression; */
OUT : AVN_PRINT ID SEMI
    | AVN_PRINT E SEMI
    ;

/* Expression → E op E | (E) | ID | NUM | STRING */
E   : E OP_ADD E
    | E OP_SUB E
    | E OP_MUL E
    | E OP_DIV E
    | LPAREN E RPAREN
    | ID
    | NUM
    | STRING
    ;
```

Special Symbols

Symbol	Meaning	Example
`|`	OR - multiple alternative productions	`S : D | A` means S can be D or A
`/* epsilon */`	Empty production - rule produces nothing	`SL : | /* epsilon */`
`{ }`	Semantic action - C code to execute	`{ printf("Success"); }`
`$$`	Result of current rule	`$$ = $1 + $2;`
`$1, $2, ...`	Values of symbols in production	`$1` = first symbol

Section 4: Epilogue (Lines 97-112) - User Functions

Error Handler Implementation

```
void yyerror(const char *s)
{
    printf(
```

```
        "Syntax Error at line %d: %s, found '%s'\n",
        yylineno,    // Line number from scanner
        s,           // Error message from Bison
        yytext       // Problematic token from scanner
    );
}
```

When called: - Automatically invoked by parser when syntax error detected - Example output: `Syntax Error at line 5: syntax error, found 'avn_int'`

Uses: - **`yylineno`**: Current line number (from scanner) - **`yytext`**: The token that caused the error (from scanner) - **`s`**: Error description from Bison (e.g., "syntax error", "unexpected token")

Common error messages: - `"syntax error"` - General parsing error - `"unexpected TOKEN"` - Wrong token at this position - `"expecting TOKEN"` - Parser was expecting a specific token

Main Function

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

`yyparse()` function: - **The main parser function** - auto-generated by Bison - **What it does:** 1. Initializes parsing state machine 2. Repeatedly calls `yylex()` to get tokens 3. Applies grammar rules (shift/reduce operations) 4. Builds parse tree (implicitly or explicitly) 5. Executes semantic actions in `{ }` blocks 6. Returns 0 on success, non-zero on error

Return values: - `0` - Parsing successful - `1` - Parsing failed (syntax error) - `2` - Memory exhaustion

---

Scanner-Parser Connection

The Complete Token Flow Pipeline

```
Source Code: "avn_int x;"




Scanner (lex.yy.c)
- Reads input character by character
```

```
- Matches patterns using regex
- Generates tokens
- Updates yytext & yylineno

            returns: AVN_INT (258)
            yytext = "avn_int"
            yylineno = 1


yylex() - THE BRIDGE FUNCTION
- Called by parser
- Implemented by scanner
- Returns integer token codes




Parser (y.tab.c)
- Receives token: AVN_INT
- Checks grammar rules
- Shifts/Reduces using LR parsing
- Builds parse tree
- Executes semantic actions




Parse Tree / AST
  Success - "Syntax analysis successful"
  Error - yyerror() called
```

Step-by-Step Connection

Step 1: Scanner includes parser's header

```
// In scanner.l (line 3):
#include "y.tab.h"
```

Purpose: - Gets token code definitions (AVN_INT = 258, etc.) - Without this, scanner wouldn't know what integers to return

Step 2: Parser generates token codes When you run `bison -d parser.y`, it creates `y.tab.h`:

```
// In y.tab.h (auto-generated):
#ifndef YYTOKENTYPE
```

```
# define YYTOKENTYPE
enum yytokentype {
    AVN_INT = 258,
    AVN_IF = 259,
    AVN_ELSE = 260,
    AVN_WHILE = 261,
    ID = 262,
    NUM = 263,
    // ... etc
};
#endif
```

Why start at 258? - Values 0-255 are reserved for single-character tokens - ASCII values directly represent themselves - Example: `'{'` = 123, `'}'` = 125

Step 3: Scanner returns these codes

```
// In scanner.l:
avn_int     { print_token("KEYWORD", yytext); return AVN_INT; }
//                                             ^^^^^^^^^^^^^^
//                                            Returns 258

avn_if      { print_token("KEYWORD", yytext); return AVN_IF; }
//                                             ^^^^^^^^^^^^^
//                                            Returns 259

{ID}        { print_token("IDENTIFIER", yytext); return ID; }
//                                               ^^^^^^^^^^
//                                              Returns 262
```

What happens internally: 1. Scanner matches pattern "avn_int" 2. Sets `yytext = "avn_int"` 3. Returns integer `258` 4. Parser receives `258` and knows it's `AVN_INT`

Step 4: Parser calls yylex() to get tokens

```
// In parser.y:
int yylex(void);  // Declaration

// Inside yyparse() (auto-generated in y.tab.c):
while (1) {
    yychar = yylex();  // Get next token
    // ... process token based on current state
}
```

The loop: 1. Parser is in some state (e.g., "expecting statement") 2. Calls `yylex()` to get next token 3. Scanner reads input, matches pattern, returns

token code 4. Parser uses state machine to decide: shift or reduce? 5. Repeat until EOF or error

Step 5: Parser uses tokens in grammar rules

```
// Grammar rule:
D : AVN_INT ID SEMI
```

Parser matching process: 1. Receives token `AVN_INT` (258) → Shift onto stack 2. Receives token `ID` (262) → Shift onto stack 3. Receives token `SEMI` (263) → Shift onto stack 4. **Pattern matches rule D!** → Reduce stack to `D` 5. Execute semantic action (if any) 6. Continue parsing

Shared Variables Between Scanner and Parser

Variable	Type	Defined In	Used In	Purpose
yylineno	`int`	Scanner (lex.yy.c)	Parser (yyerror)	Track current line number
yytext	`char*`	Scanner (lex.yy.c)	Parser (yyerror)	Current token text
yyleng	`int`	Scanner (lex.yy.c)	Parser (optional)	Length of yytext
Token codes	`enum`	Parser (y.tab.h)	Scanner (return)	Token identification

Communication Diagram

```
scanner.l                    parser.y
    |                            |
    | #include "y.tab.h"         |
    |<---------------------------| (provides token codes)
    |                            |
    | return AVN_INT;            |
    |--------------------------->| (sends tokens)
    |                            |
    | yytext = "avn_int"         |
    |--------------------------->| (shared variable)
    |                            |
    | yylineno = 5               |
    |--------------------------->| (shared variable)
    |                            |
    |      yylex() call          |
    |<---------------------------| (requests next token)
    |                            |
```

Generated Files

Compilation Process

```
# Step 1: Generate scanner from scanner.l
flex scanner.l
# Output: lex.yy.c (C code for lexical analyzer)

# Step 2: Generate parser from parser.y
bison -d parser.y
# Output:
#   y.tab.c (C code for parser)
#   y.tab.h (Token definitions)

# Step 3: Compile everything together
gcc lex.yy.c y.tab.c -o avian_compiler

# Step 4: Run the compiler
./avian_compiler < input.avn
```

y.tab.h - Header File

Purpose: Token definitions shared between scanner and parser

File size: ~100-150 lines

Contains:

```
/* 1. Token type enumeration */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
enum yytokentype {
    AVN_INT = 258,
    AVN_IF = 259,
    AVN_ELSE = 260,
    // ... all tokens
};
#endif

/* 2. Token type codes as macros (for compatibility) */
#define AVN_INT 258
#define AVN_IF 259
// ... etc

/* 3. Parser function declaration */
int yyparse(void);

/* 4. Token value type (if using %union) */
```

```c
#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE {
    int intval;
    float floatval;
    char* strval;
} YYSTYPE;
# define YYSTYPE_IS_DECLARED 1
#endif

/* 5. External declarations */
extern YYSTYPE yylval;
```

Who includes this file: - `scanner.l` - Needs token codes - Any other C file that wants to call the parser

y.tab.c - Implementation File

Purpose: The complete parser implementation in C

File size: ~1500-2000 lines (machine-generated)

Contains:

1. Parse Tables (LR State Machine)

```c
/* Parse action table - what to do in each state */
static const yytype_int16 yypact[] = {
    -7,   -7,   12,   -6,  -11,   15,   16,  ...
};

/* Default actions for each state */
static const yytype_int8 yydefact[] = {
    0,    2,    0,    0,    0,    1,    3,  ...
};

/* State transition table */
static const yytype_int8 yytable[] = {
    4,    5,    6,    1,    7,    8,    9,  ...
};

/* Table verification/checking */
static const yytype_int8 yycheck[] = {
    0,    1,    2,    0,    4,    5,    6,  ...
};
```

2. Grammar Rule Encoding

```c
/* Left-hand side of each rule (which non-terminal) */
static const yytype_int8 yyr1[] = {
    0,   47,   48,   49,   50,   50,   51,  ...
};

/* Length of right-hand side (how many symbols) */
static const yytype_int8 yyr2[] = {
    0,    1,    3,    2,    0,    1,    1,  ...
};
```

3. The Main `yyparse()` Function

```c
int yyparse(void)
{
    int yystate = 0;
    int yyerrstatus = 0;
    int yychar = YYEMPTY;

    /* State stack */
    yytype_int16 yyssa[YYINITDEPTH];
    yytype_int16 *yyss = yyssa;
    yytype_int16 *yyssp = yyss;

    /* Main parsing loop */
    while (1) {
        /* Get next token if needed */
        if (yychar == YYEMPTY) {
            yychar = yylex();
        }

        /* Determine action based on current state and token */
        yyn = yypact[yystate];

        /* Shift or reduce? */
        if (yyn > 0) {
            /* Shift: push token onto stack */
            *++yyssp = yystate = yyn;
            *++yyvsp = yylval;
            yychar = YYEMPTY;
        } else {
            /* Reduce: apply grammar rule */
            yyn = yyr1[yyn];
            // ... reduction logic
        }
    }
}
```

4. Your Semantic Actions

```
switch (yyn)
{
    case 1:  /* P -> B */
#line 49 "parser.y"
    {
        printf("Syntax analysis successful\n");
    }
#line 1234 "y.tab.c"
    break;

    case 2:  /* B -> LBRACE SL RBRACE */
#line 55 "parser.y"
    {
        /* semantic action if any */
    }
#line 1243 "y.tab.c"
    break;

    // ... more cases
}
```

5. Token Name Table (for debugging)

```
static const char *const yytname[] = {
    "$end", "error", "$undefined",
    "AVN_INT", "AVN_IF", "AVN_ELSE",
    "AVN_WHILE", "AVN_PRINT", "ID",
    "NUM", "STRING", "ASSIGN",
    // ... all token names
};
```

Key Differences

Aspect	y.tab.h	y.tab.c
Type	Header file	Implementation file
Purpose	Declarations	Definitions
Size	~100 lines	~1500-2000 lines
Contains	Token codes, prototypes	Parser logic, state tables
Included by	scanner.l, other files	Compiled separately
Readability	Human-readable	Machine-generated, complex
Edit	Never (auto-generated)	Never (auto-generated)
Version control	Usually excluded	Usually excluded

y.output - Parser Analysis (Optional)

Generated with `bison -v parser.y`:

```
State 0

    0 $accept: . P $end

    LBRACE  shift, and go to state 1

    P  go to state 2
    B  go to state 3

State 1

    1 B: LBRACE . SL RBRACE

    AVN_INT    shift, and go to state 4
    AVN_IF     shift, and go to state 5
    ...
```

Purpose: - Debug shift/reduce and reduce/reduce conflicts - Understand state machine behavior - Not needed for compilation

---

Token Flow Example

Input Code:

```
{
    avn_int x;
    x = 5;
}
```

Scanner Processing:

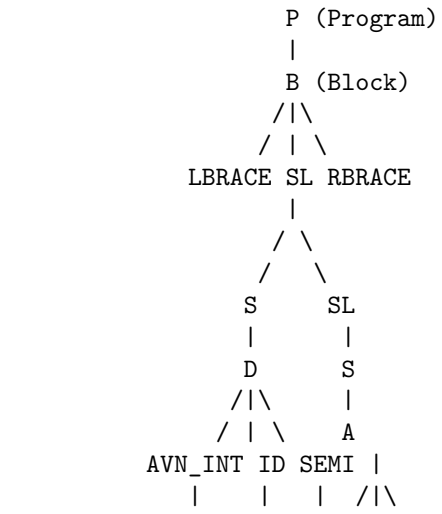
Step	Input	Pattern Match	Token Returned	Token Code	yytext Value
1	`{`	`"{"`	LBRACE	123	`"{"`
2	`avn_int`	`avn_int`	AVN_INT	258	`"avn_int"`
3	`x`	`{ID}`	ID	262	`"x"`
4	`;`	`";"`	SEMI	263	`";"`
5	`x`	`{ID}`	ID	262	`"x"`
6	`=`	`"="`	ASSIGN	264	`"="`
7	`5`	`{INT}`	NUM	265	`"5"`
8	`;`	`";"`	SEMI	263	`";"`

Step	Input	Pattern Match	Token Returned	Token Code	yytext Value
9	}	"}"	RBRACE	125	"}"

Parser Processing (Bottom-Up LR Parsing):

```
Stack                    Input                  Action
-----                    -----                  ------
$                        { avn_int x; x = 5; }  shift {
$ {                      avn_int x; x = 5; }    shift avn_int
$ { avn_int              x; x = 5; }            shift x
$ { avn_int x            ; x = 5; }             shift ;
$ { avn_int x ;          x = 5; }               reduce by D -> avn_int x ;
$ { D                    x = 5; }               reduce by S -> D
$ { S                    x = 5; }               shift x
$ { S x                  = 5; }                 shift =
$ { S x =                5; }                   shift 5
$ { S x = 5              ; }                    reduce by E -> 5
$ { S x = E              ; }                    shift ;
$ { S x = E ;            }                      reduce by A -> x = E ;
$ { S A                  }                      reduce by S -> A
$ { S S                  }                      reduce by SL -> S SL
$ { SL                   }                      shift }
$ { SL }                                        reduce by B -> { SL }
$ B                                             reduce by P -> B
$ P                                             ACCEPT
```

Parse Tree Built:



```
"avn_int" "x" ";" / | \
                  ID ASSIGN E SEMI
                  |    |    |  |
                 "x"  "="  NUM ";"
                            |
                           "5"
```

Execution Flow:

```
1. main() is called
2. main() calls yyparse()
3. yyparse() enters main loop:

   Loop iteration 1:
   - Calls yylex() → returns LBRACE (123)
   - Shifts LBRACE onto stack

   Loop iteration 2:
   - Calls yylex() → returns AVN_INT (258)
   - Shifts AVN_INT onto stack

   Loop iteration 3:
   - Calls yylex() → returns ID (262)
   - Shifts ID onto stack

   Loop iteration 4:
   - Calls yylex() → returns SEMI (263)
   - Shifts SEMI onto stack
   - Recognizes pattern: AVN_INT ID SEMI
   - Reduces to D
   - Reduces to S

   Loop iteration 5:
   - Calls yylex() → returns ID (262)
   - Shifts ID onto stack

   ... continues until EOF or error ...

   Final iteration:
   - All input consumed
   - Stack contains: P (Program)
   - Executes semantic action: printf("Syntax analysis successful\n")
   - Returns 0 (success)
```

YY-Prefixed Elements Reference

All `yy` prefixes come from **Yacc (Yet Another Compiler Compiler)**, Bison's predecessor.

Core Functions

Function	Defined In	Called By	Returns	Purpose
yylex()	Scanner (lex.yy.c)	Parser (yyparse)	`int`	Returns next token code
yyparse()	Parser (y.tab.c)	main()	`int`	Main parsing function (0=success)
yyerror()	User (parser.y)	Parser (on error)	`void`	Reports syntax errors
yywrap()	User/Scanner	Scanner (at EOF)	`int`	Handle multiple input files (return 1 = done)

Global Variables

Variable	Type	Defined In	Set By	Purpose
yytext	`char*`	Scanner	Scanner	Points to current token text
yyleng	`int`	Scanner	Scanner	Length of current token
yylineno	`int`	Scanner	Scanner	Current line number
yylval	`YYSTYPE`	Parser	Scanner	Semantic value of current token
yychar	`int`	Parser	Parser	Current lookahead token
yydebug	`int`	Parser	User	Enable debug output (set to 1)

Variable	Type	Defined In	Set By	Purpose
yynerrs	`int`	Parser	Parser	Number of errors encountered

Type Definitions

Type	Purpose	Example
YYSTYPE	Union type for token/rule values	`union { int i; char* s; }`
YYLTYPE	Type for location tracking	`struct { int first_line; ... }`

Macros and Constants

Name	Value	Purpose
YYEMPTY	-2	No lookahead token yet
YYEOF	0	End of file token
YYerror	256	Error token
YYACCEPT	-	Macro to accept successfully
YYABORT	-	Macro to abort parsing

Generated Tables (in y.tab.c)

Table	Type	Purpose
yypact[]	`int16[]`	Parse action table (shift/reduce decisions)
yydefact[]	`int8[]`	Default action for each state
yypgoto[]	`int16[]`	Goto table for non-terminals
yydefgoto[]	`int16[]`	Default goto for each non-terminal
yytable[]	`int16[]`	State transition table
yycheck[]	`int8[]`	Verification table
yyr1[]	`int8[]`	Left-hand side of each grammar rule
yyr2[]	`int8[]`	Length of right-hand side of each rule
yytname[]	`char*[]`	Token/non-terminal names (for debugging)

Debug Functions

```c
extern int yydebug;  // Debug flag

int main(void) {
    yydebug = 1;     // Enable verbose parsing output
    return yyparse();
}
```

Debug output shows:

```
Starting parse
Entering state 0
Reading a token: Next token is token LBRACE ()
Shifting token LBRACE ()
Entering state 1
Reading a token: Next token is token AVN_INT ()
Shifting token AVN_INT ()
...
Reducing stack by rule 3 (line 69):
   $1 = token AVN_INT ()
   $2 = token ID ()
   $3 = token SEMI ()
-> $$ = nterm D ()
...
```

Common Patterns Explained

1. Left Recursion (Efficient for Bottom-Up Parsers)

```
E : E OP_ADD E    /* Left-recursive */
  | NUM
  ;
```

Why used: - More efficient for LR/LALR parsers like Bison - Uses less stack space - Parses `1 + 2 + 3` as `((1 + 2) + 3)` - left-associative

Stack behavior for `1 + 2 + 3`:

```
1. Shift 1, reduce to E
2. Shift +
3. Shift 2, reduce to E
4. Reduce: E + E -> E    (now have: 1+2)
5. Shift +
6. Shift 3, reduce to E
7. Reduce: E + E -> E    (now have: (1+2)+3)
```

2. Right Recursion (For Top-Down Parsers)

```
SL : S SL          /* Right-recursive */
   | /* empty */
   ;
```

When to use: - When order matters - For list-building - In this case: processes statements left-to-right

Note: Bison handles both efficiently, but left-recursion is preferred when possible.

3. Epsilon (Empty) Productions

```
SL : S SL
   | /* epsilon */
   ;
```

Meaning: - StatementList can be empty - Allows `{ }` (empty block) to be valid - Base case for recursion

Without epsilon:

```
SL : S SL
   | S     /* At least one statement required */
   ;
```

This would require `{ avn_int x; }` but disallow `{ }`.

4. Multiple Alternatives with |

```
S : D
  | A
  | IF
  | WH
  | OUT
  ;
```

Equivalent to:

```
S : D ;
S : A ;
S : IF ;
S : WH ;
S : OUT ;
```

5. Semantic Actions with $$ and $n

```
E : E OP_ADD E
    { $$ = $1 + $3; }  /* $$ = result, $1 = first E, $3 = third symbol (second E) */
  | NUM
```

```
    { $$ = $1; }        /* $$ = result, $1 = NUM value */
  ;
```

Symbol numbering: - `$$` - The result (left-hand side) - `$1` - First symbol on right-hand side - `$2` - Second symbol - `$3` - Third symbol - etc.

Error Recovery

Basic Error Recovery

Bison provides a special `error` token for error recovery:

```
S : D
  | A
  | IF
  | WH
  | OUT
  | error SEMI      /* Skip to next semicolon on error */
  ;
```

How it works: 1. Parser encounters error 2. Calls `yyerror()` 3. Pops stack until it finds a state that accepts `error` token 4. Discards input tokens until synchronization point (SEMI) 5. Resumes parsing

Advanced Error Recovery

```
statement_list
  : statement
  | statement_list statement
  | error ';'       { yyerrok; yyclearin; }
  ;
```

Special actions: - `yyerrok;` - Reset error count, stop error recovery mode - `yyclearin;` - Discard current lookahead token

---

Practical Examples

Example 1: Valid Avian Program

```
{
    avn_int x;
    x = 10;
    avn_print x;
}
```

Parse trace: 1. LBRACE → shift 2. AVN_INT → shift 3. ID(“x”) → shift 4. SEMI → shift, reduce to D, reduce to S 5. ID(“x”) → shift 6. ASSIGN → shift

7. NUM(10) → shift, reduce to E 8. SEMI → shift, reduce to A, reduce to S 9. AVN_PRINT → shift 10. ID("x") → shift, reduce to E 11. SEMI → shift, reduce to OUT, reduce to S 12. Build SL from statements 13. RBRACE → shift, reduce to B, reduce to P 14. **Success!** Prints: "Syntax analysis successful"

Example 2: Syntax Error

```
{
    avn_int x
    x = 10;
}
```

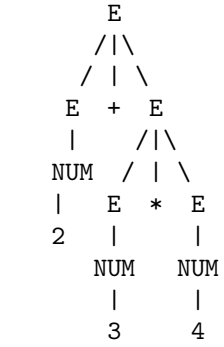
Error: Missing semicolon after `avn_int x`

Parser behavior: 1. Processes: LBRACE, AVN_INT, ID("x") 2. Expects SEMI but receives ID("x") 3. Calls `yyerror("syntax error")` 4. Prints: `Syntax Error at line 2: syntax error, found 'x'` 5. Returns 1 (failure)

Example 3: Expression Precedence

```
{
    avn_int result;
    result = 2 _+_ 3 _*_ 4;
}
```

Parse tree for `2 _+_ 3 _*_ 4`:



Evaluates as: `2 + (3 * 4) = 14`

Because `%left OP_MUL OP_DIV` appears below `%left OP_ADD OP_SUB`, giving it higher precedence.

Debugging Tips

1. Enable Parser Debug Output

```
int main(void) {
    yydebug = 1;  // Verbose output
    return yyparse();
}
```

2. Generate State Machine Description

```
bison -v parser.y
# Creates y.output with detailed state information
```

3. Add Print Statements in Semantic Actions

```
D : AVN_INT ID SEMI
    { printf("Declared variable: %s\n", $2); }
  ;
```

4. Check for Conflicts

```
bison -v parser.y
# Look for shift/reduce or reduce/reduce conflicts in output
```

Warning messages:

```
parser.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
parser.y: warning: 2 reduce/reduce conflicts [-Wconflicts-rr]
```

5. Common Errors

Error Message	Cause	Solution
`syntax error, unexpected ID`	Wrong token received	Check grammar rules
`undefined reference to yylex`	Scanner not compiled	Link lex.yy.c
`yytext undeclared`	Missing extern declaration	Add `extern char* yytext;`
`shift/reduce conflict`	Ambiguous grammar	Add precedence rules or rewrite grammar

Advanced Features (Not Used in This Parser)

1. Semantic Values with %union

```
%union {
    int intval;
    float floatval;
    char* strval;
}

%token <intval> NUM
%token <strval> ID
%type <intval> E

E : NUM         { $$ = $1; }
  | E '+' E     { $$ = $1 + $3; }
  ;
```

2. Location Tracking with @n

```
%locations

E : E '+' E
    {
        printf("Addition at line %d\n", @2.first_line);
        $$ = $1 + $3;
    }
  ;
```

3. Pure (Reentrant) Parser

```
%define api.pure full

int yylex(YYSTYPE *yylval_param, void *scanner);
int yyparse(void *scanner);
```

Quick Reference Card

Bison Directives

Directive	Purpose	Example
`%token`	Declare terminal symbol	`%token AVN_INT`
`%start`	Define start symbol	`%start program`
`%left`	Left-associative operator	`%left '+' '-'`
`%right`	Right-associative operator	`%right '^'`

Directive	Purpose	Example
`%nonassoc`	Non-associative operator	`%nonassoc '<' '>'`
`%union`	Define semantic value types	`%union { int i; }`
`%type`	Declare non-terminal type	`%type <i> expr`
`%prec`	Override precedence	`expr: '-' expr %prec UMINUS`

Special Symbols in Grammar

Symbol	Meaning
`|`	Alternative production
`:`	"Produces" / "Is defined as"
`;`	End of rule
`{ }`	Semantic action (C code)
`$$`	Value of current rule
`$1, $2, ...`	Values of symbols in production
`@$`	Location of current rule
`@1, @2, ...`	Locations of symbols

Compilation Commands

```
# Generate parser with verbose output
bison -d -v parser.y

# Generate scanner
flex scanner.l

# Compile
gcc lex.yy.c y.tab.c -o compiler

# Run
./compiler < input.avn
```

Resources

- **Bison Manual**: https://www.gnu.org/software/bison/manual/
- **Flex Manual**: https://github.com/westes/flex
- **Dragon Book**: "Compilers: Principles, Techniques, and Tools" by Aho et al.
- **Engineering a Compiler**: Cooper & Torczon

Summary

The key takeaways:

1. **parser.y defines the syntax** of your language using grammar rules
2. **scanner.l defines the tokens** that parser.y uses
3. **yylex() is THE BRIDGE** - parser calls it, scanner implements it
4. **Token codes** flow from parser → scanner → back to parser
5. **Shared variables** (`yytext`, `yylineno`) enable error reporting
6. **y.tab.h** contains token definitions shared between files
7. **y.tab.c** contains the complete parser implementation
8. **Parse tree is built bottom-up** using shift/reduce operations

The complete flow:

```
Source → Scanner → Tokens → Parser → Parse Tree → Success/Error
         (lex.yy.c)         (y.tab.c)
```

Last Updated: January 22, 2026
Author: Nayab Irfan