Compiler Construction Review.

Introduction Slides

1. Our Languages.

SLIC	Source lang.
GSTAL	target lang.
C	primary lang. for coding.
Flex	lang ~~fo~~ for writing ~~a~~ scanner
Bison	lang. for writing parser
Make.	lang for managing the project.

SLIC ⟶ our compiler ⟶ GSTAL.

2. Language:
   a set of strings. (finite or infinite)

3. 2 Aspects of a Prog. Lang.
   ① Syntax : form / structure

   ② Semantics : meaning

4. Func of a compiler:
   translate str. of one lang. (source) into str. of another lang. (target) in such a way that the semantics ~~is p~~ are preserved.

5. Typical Compiation Process.

Source Prog
↓ characters

Analysis Phase (Front End)

Lexical Analyzer – recognize tokens / word-level constructs.
↓ tokens
Syntax Analyzer – recognize phrase-level constructs.
↓ syntax tree, symbol table
Semantic Analyzer – enforce data type rules.
↓ decorated syntax tree
Intermediate Code Generator – produce code in machine-independent lang.
↓ intermediate code

Synthesis phase (Backend)

Machine-Independent Code ~~Generator~~ Optimizer – optimize machine-independent code.
↓ modified intermediate code
Code Generator – produce machine code
↓ target machine code.
Machine-Dependent Code Optimizer – optimize machine code.
↓ modified target machine code.
Target Prog.

6. Our Compiler

Slic Prog
↓ characters
Lexical Analyzer / Scanner
↓ tokens
Syntax Analyzer / Parser
↓ ST & AST.
Code Generator
↓ GSTAL code.
GSTAL Prog.

Front End. Product: ST & AST.

BackEnd: GSTAL code.

Regular Lang. & Regex

1. Chomsky's Hierarchy

Type 0	Recursively Enumerable Lang.	
1	Context - sensitive Lang.	
2	Context - free Lang.	→ parser
3	Regular Lang.	→ scanner

2. 3 Fundamental Operations

① Concatenation — via juxtaposition
↳ sequence

② alternation — |
↳ selection — ~~\|~~

③ ~~Klt~~ Kleene Closure — *
↳ repetition

(positive closure +).

3. Flex Regex :

[ ] character class. eg. [0-9] , [a-z] , [0-9a-zA-Z] .

^ match everything except. eg [^\n] .

• any char except \n

{ , } ~~Ⅱ~~ fill in 1 or 2 no. ~~=~~ min & max. eg. A{1,3} = {AAA, AA, A}
0{5} = 00000

\ escape. eg. \"

* match 0^+ occurrence. eg. [\t]*

+ match 1^+ occurrence eg. [0-9]+

? match 0 ~~or~~ or 1. eg. (+|-)? [0-9]+
⇑
can be present or not.

Context Tree Lang & CFG

1. BNF · Bactus-Naur Form.

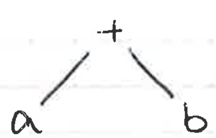
eg. expression CFG · left recursion ·

expr → expr + term
expr → expr − term
expr → term
term → term * factor
term → term / factor
term → factor
factor → (expr)
factor → ~~atto~~ ATOM.

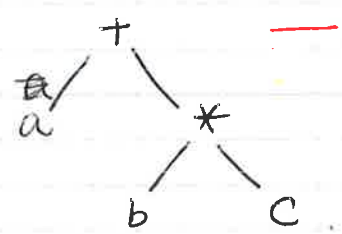
terminals in capitalized letters.

Expression Tree

a + b



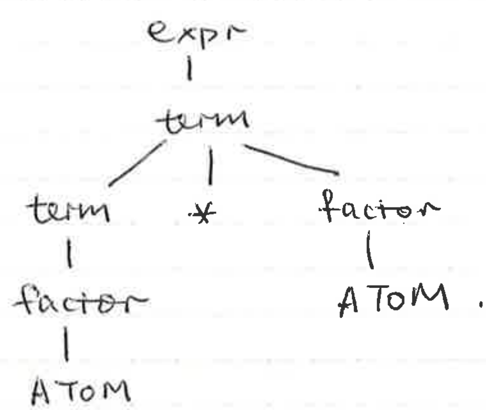
a + b * c



→ last op. higher in tree.
left recursion gives op. associativity.

2. Parse Trees. v. Derivations

eg. ATOM * ATOM.



expr ⇒ ~~exp~~ term
⇒ term * factor
⇒ factor * factor
⇒ ATOM * factor
⇒ ATOM * ATOM.

3. Ambiguous Grammars:

If there ~~e~~ exist one string in the lang. that can be derived from 2 dif' left most derivations, or 2 dif' parse trees → ambiguous.

$\boxed{\text{GSTAL}}$

1. GSTAL Virtual Machine:

✰ zero-address machine ✰.

- The machine instructions do not include memory addresses.
- They retrieve operands from a central stack & push results onto same stack.

2. GSTAL Memory Architecture:

✰ ① Harvard Architecture:
code & data memory memory are separate.
eg. in microcontrollers.

* ② von Neumann Architecture:
(not GSTAL) code & data share the same memory space
eg. personal computers.

3. Code Memory.

① each location holds 1 GSTAL instruction.

② 1^{st} instruction starts at addr. Ø.

③ no inherent upper bound on limit.
(depend on size of the process that the GSTAL interpreter runs).

4. Data Memory.

① an array of 4-byte words, addressable by word (32-bit)
NOT byte-(8-bit)

② Each memory word :
1) int
2) float
~~3) GSTAL instruction~~

③ The stack is initially empty.

④ Violations allowed :
to store & retrieve variables.
eg. ISP #.

⑤ Any addr ref. < 0 or > tos
results in Runtime Error / Execution Error.

5. 3 Registers.

⊕ registers values are altered as side effects of the various GSTAL instructions.
can't be changed directly.

① tos - top of stack.
addr. in data memory of the current top entry of the stack
⊖ addr. ref > tos or < 0 → execution error.
TOS undefined if empty stack

② pc - program counter.
addr. in code memory of the current GSTAL instruction. initial at zero.

③ act - base addr. in data memory of the current activation frame. * relevant to subroutine calls, returns, & parameter passing.

6. Comments & Blank Lines.

✱ Every line must have an instruction. ✱

comment :
instr. ; followed by any text till EOL.

eg. HLT ; here is comment.

7. GSTAL Interpreter.

b/f execution:

scan entire prog. to verify syntax.

① If syntax error:
→ report error &
~~to~~ abort the run

② else:
run the prog.

Termination under 3 conditions:

① HLT

② physically the last statement is executed
* NOT a JMP, JPF, RET * .
→ fall thru the bottom.

③ Execution Error :
report error & terminate.
eg. addr. ref. outside range.

```
$ cd gstal
$ make
```

```
$ ./gstal filename.gstal
```

```
$ ./gstal -d file.gstal
```

run prog. & produce a stack dump if execution error occurs.

```
$ ./gstal -l file.gstal.
```

not run prog, write a numbered listing of the prog to the standard output.

$\boxed{\text{SLIC}}$ imperative lang. ~ C.

1. Rules of Syntax.

a prog only consist of a main prog.
(NO subroutines / func).

```
main;
   ~~~~~~
end main;
```

① Each statement ends w/ ;

② Reserved words NOT case-sensitive.
main, end, exit, if, else, while, to, counting, upward, downward, real, integer, data, algorithm, read, print.

③ Var names & other user defined identifiers.
→ case-sensitive.

④ data & alg section.

```
main;
    data:
        ~~~~~~    ;
    algorithm:
        ~~~~~~    ;
end main;
```

⑤ Comment begin w/ #. & till EOL.

2. Terminating Execution

① exit statement

exit ;

② appear anywhere in alg. section ✓

③ prog can have any no. of exit statements ✓

3. Variables, Data Types, Declarations.

① 2 Data Types. { integer / real

* when a decimal pt. is present, there must be at least 1 digit on each side of the pt.

② var names: alphanumeric & begin w/ letter.
case-sensitive.

③ Declaration:

```
data :
    integer : a, b, c ;
    integer : X ;
    real    : m, n, fido ;
    real    : fluffy [10] ;
```

⇒ fluffy [0] thru fluffy [19]

[10] ↑ nonnegative int. (no. of ele.)

④ one declaration stmt can declare any no. of var.
data sec. can have any no. of declarations.

④ Boolean.

no boolean data type.

zero → false

non-zero → true.

0 - F

1 - T

4. Expressions :

① infix notation .

② contain : integer & real constants
variables
operators
parentheses .

* nonint. array subscript → coerce to int (truncation) .

③ Arithematic Op : +, −, *, /, %(& unary minus).

1) Division : /

2 int operands → int result .
if 1 real operand → real

2) Mod : % .

2 int operands → int .

* real coerced to int

④ Relational Op : >, <, ≥, ≤, <>, = .

0 − F
1 − T

⑤ Boolean Logic Op : &, |, ~

⑥ Op. Priority :

highest → 1) unary minus −
2) *, /, % .
3) +, −
4) <, >, ≤, ≥, =, <>
5) &, |, ~

5. Assignment Stmts.

varref := expr ;

① varref : → target

variable name (scalar)

array ref. w/ subscript in range.

② expr : → source.

valid expression.

③ Mixed-mode assignments :

✱ coerce to appropriate data type ✱.

1) int val → real var.

coerce int to real

2) real val → int var

truncate to coerce real to int.

6. Repetition Control Structures

① Conditional loop (while).

while expr ;

~~~~ ;

end while ;

1) expr evaluated b/f each loop iteration.

= T → run

= F / 0 → skip.

② Counting loops.

counting var upward expr1 to expr2 ;

~~~ ;

end counting ;

counting var downward expr1 to expr2 ;

~~~ ;

end counting ;
~~~

1) expr 1 . evaluate & store in var's addr.
(start) each iteration. compare w/ expr2.
b/f loop begin :
var is initialized to value of expr 1.

2) expr 2 . lock in value when evaluate 1st time
(end) → synthesize a var declaration &
store value

3) var → integer scalar var
expr1 & expr2 → int expressions.

4) upward. var += 1 after each loop iteration
downward. var -= 1.

7. Selection Ctrl Structures

```
if expr ;
    ~~~ ;
end if;
```

```
if expr ;
    ~~ ;
else ;
    ~~ ;
end if;
```

8. I/O Stmts

① read Stmts.

read varref;

when executed, prog. pause & wait for keyboard input. * one varref only each stmt *.

② print Stmts.

print printlist;

★ Printlist ★

a comma-delimited list of print items :

1) an expr.

2) char str. in double quotes . " ~~ "

"" to ~~esq~~ escape & print 1 "

* must be in one line * .

3) exclamation mark ! newline / carriage return .

```
print " Sum is ", Sum , ! ;
print " Value is :   ", Val [0] , ! ;
print " He said "" Hello "" to me ", ! ;
```

Textbook Notes

Ch 1. Intro

1. Lexical Analysis / Scanning :
   tokens : atomic vals, smallest indivisible word-level construct.

2. Syntax Analysis / Parsing :
   phrase-level construct.

3. Flex Prog.
   ⊕ Flex translates all the regex into an efficient internal form & let it match the input against all the patterns simultaneously.

Ex. word count .l

```
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
```

declarations & option settings

%% . → delimiters.

regex

*must start at begin of line

```
[a-zA-Z]+    { words++; chars += strlen(yytext); }.
\n           { chars ++; lines ++; }.
.            { chars ++; }.
%%.
```

```
main (int argc, char ** argv)
{
  yylex();
  printf("%8d %8d %8d\n", lines, words, chars);
}
```

C code.

code sec.

⊖ flex consider any line starting w/ whitespace, code to be copied to C lexeme: the actual input ~~text~~ string that matched the regex that is currently detected by yylex().

① yytext:
the input text that the pattern just matched.

② yylex():
the name flex gives to scanner routine.

③ Compile:
$ flex scanner.l
$ gcc lex.yy.c -ll (or -lfl).
the C prog. generated.

④ Final "Trash" token:
. matches anything except \n.

4. Scanner Coroutine:
flex scanner returns a stream of tokens handled by parser (bison).

① Each time parser needs a token
→ call yylex()

② Each time scanner returns,
it remembers where it was.
If no return (in C code)
yylex() keeps going w/in same call & scanning continues immediately.

* If action code returns.
Scanning resumes on next call to yylex()

If not, scanning resumes immediately.

5. Tokens & Values

& { token → small int.
  token's value. → actual val.

* when bison creates a parser, it assigns token no. start at >58.

① yylval:
  the variable that stores the token val. ~~an int~~.

$\boxed{\text{Ex. in my prog. scanner.l}}$

```
[a-zA-Z][a-zA-Z0-9]*      { yylval.sval = strdup(yytext);
                            return (VARIABLE); } .

\"(([^"\n]*)|(\"\"))*\"   {yylval.sval = strdup(yytext)
                            return (STRING);} .

[0-9]+                    { yylval.intval = atoi (yytext);
                            return (INTCONSTANT);} .

[0-9]+ ((\.[0-9]+)|((\.[0-9]+)?[eE][-+]?[0-9]+))
                          { yylval.realval = atof(yytext);
                            return (REALCONSTANT);} .
```

6. Bison Rule (BNF Grammar).

eg. expression

```
expr   : expr + term
       | expr - term
       | term
       ;

term   : term * fact
       | term / fact
       | fact
       ;

fact   : ( expr )
       | ATOM
       ;
```

Ex. parser.y

```
%{
#include <stdio.h>
int yyerror();
int yylex();
int yyparse();
%}
```

literal code block.

```
%token ADD SUB.

%type <node> algorithm.
```

% start program

starting var.

```
%%.
```

< CFG >

```
%%
```

< C code >

Bison Advantage .

① any parser created bison has exactly 1 way to parse any input
→ NO ambiguity .
(bison reports conflicts , but will pick 1 to execute) .

② one token lookahead property .
→ can be modified to arbitrary lookahead .

Ch2 Using Flex

1. Regular Expressions :

. match any single char except \n .

[ ] char class .
if [^]
→ match any char except ones in [ ]

~~{a-z}~~
[a-z]{-}[jv] 1st class omitting 2nd class

{ } min , max .
A{1,3} . matches A , AA , AAA .
0{5} matches 00000

? zero or one occurrence .

| alternation .

Definition Section — literal code block + others
Rules Section . — regex / CFG .
User Subroutines — C code .

2. Rules when matching Regex in Bion.

① match longest possible str.
(longest str. rule).

② in case of a tie, match top one in the prog.
(top-to-bottom rule).

Ch3 Using Bison

1. flex recognize regex (scanner / lexical –)
bison recognize entire grammars. (parser / syntax –).

2. Shift / Reduce Parsing :
as parser reads tokens, each time it reads a token that doesn't complete a rule, it push the token on an internal stack & switches to a new state reflecting the token it just read. → shift.

when all symbols found to complete a rule, pop symbols off. push LHS onto stack. → reduction.

3. Bison Parser

① Definition Section :
handle control info. for the parser & set up the execution env. where the parser will operate.

② Rules for parser (BNF).

③ C code.

4. Abstract Syntax Trees (AST)

my prog AST.h

```
typedef struct Node {
        Type     dtype;
        Opkind   kind;
        struct   Node  *left;
        struct   Node  *right;
        struct   Node  *next;
        int      intval;
        float    realval;
        char     *sval;
        struct   Node  *ifelse;
        struct   Node  *cnt1;
        struct   Node  *cnt2;
} Node;
```

% union construct :
declare types to be used in the values of symbols in the parser.

Ch6 Bison Specs

1. Action :
a C code executed when bison matches a rule in grammar

⊕ The action can refer to the values associated w/ symbols in the rule by using $#. eg. 1st symbol after : is 1. $1.

$ ↑ var reference · $$ refers to the value for LHS symbol. (left of colon).

{ $$ = $1; } default in bison for rules w/ no action.

2. yyerror()

my prog. parse.y

```
int yyerror( const char * msg ){
    printf("%s \n", msg);

    printf("Called yyerror()\n");

    return Ø;                    }.
```

3. yyparse().

* entry pt. to a bison generated parser *.

When a prog. calls yyparse():
the parser attempts to parse an input stream.

Return value int = Ø if succeed

≠ 0 if fail

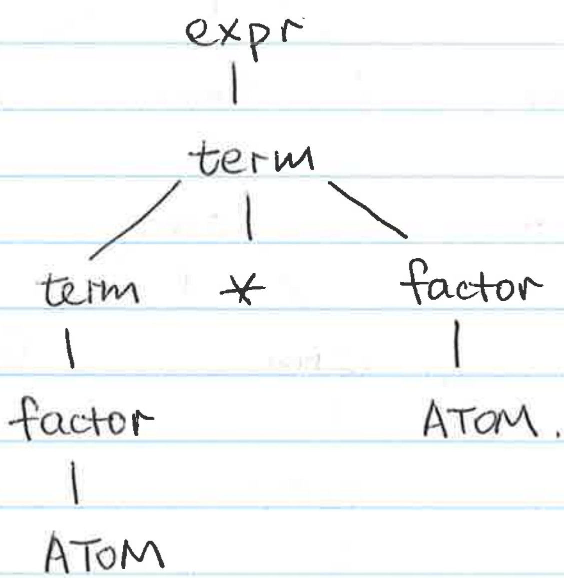
scanner.l
```
int main {
    yylex();
    return 0;
}
```

parser.y
```
int yyerror() {
    printf(" Called yyerror() \n");
    return 0;
}
```

10

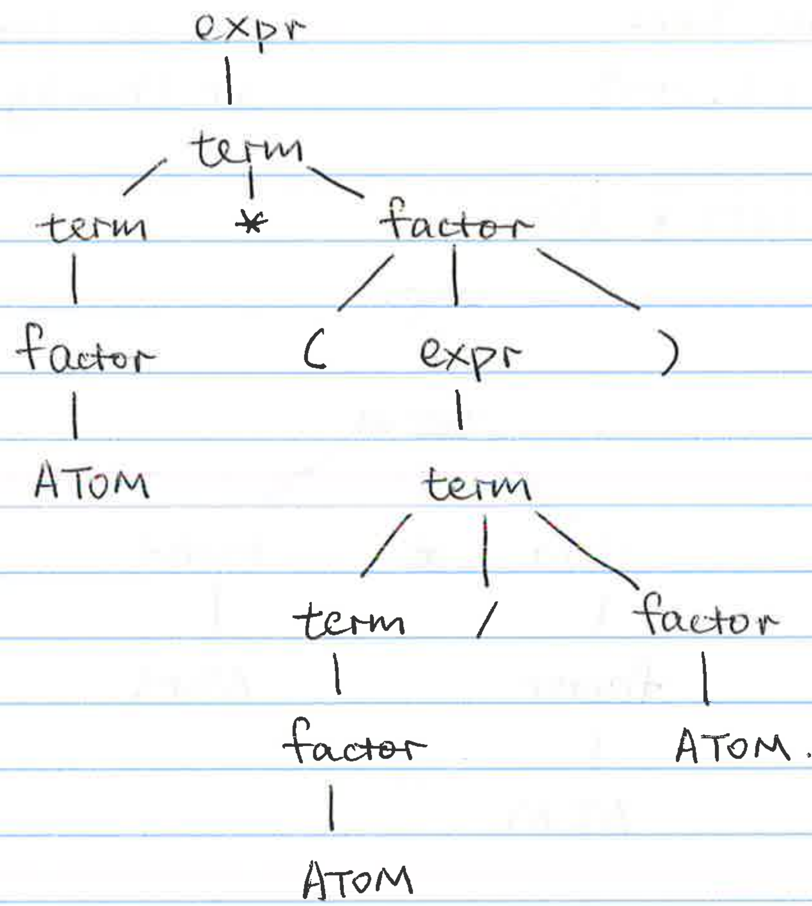
Cecilia Y. Sui. Homework #4
Parse Trees. Compiler Construction
Oct 1, 2019. Dr. Crawley.

1. ATOM * ATOM .



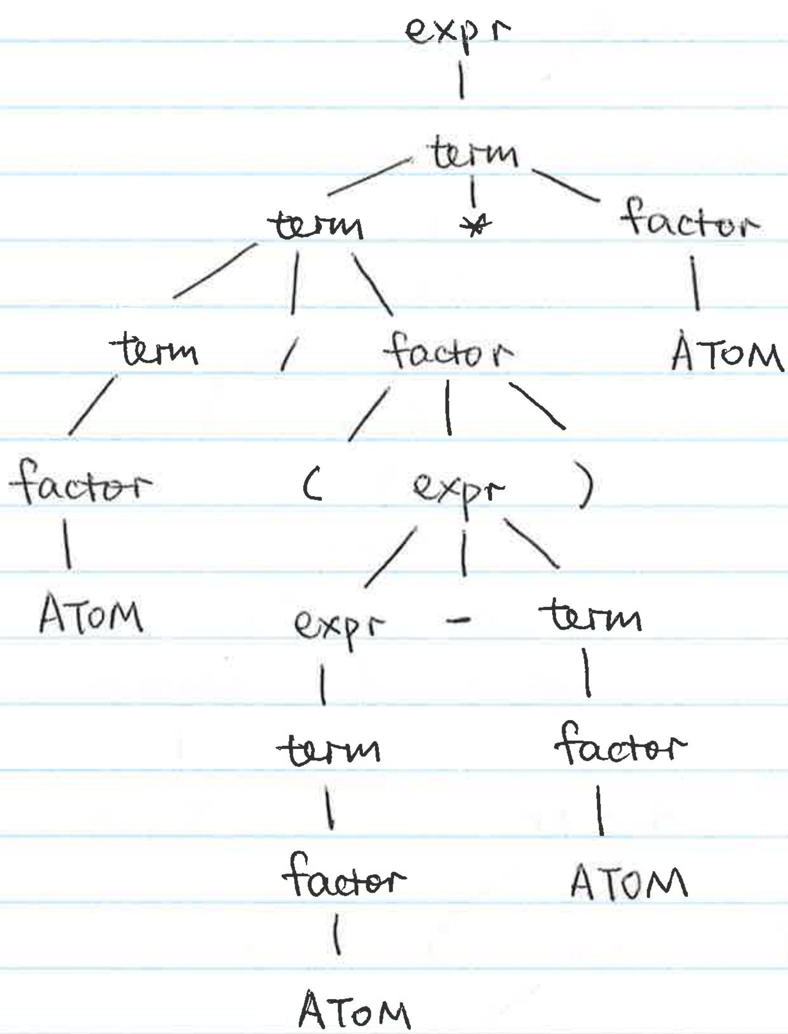
Cecilia Sui.

2. ATOM * (ATOM / ATOM)



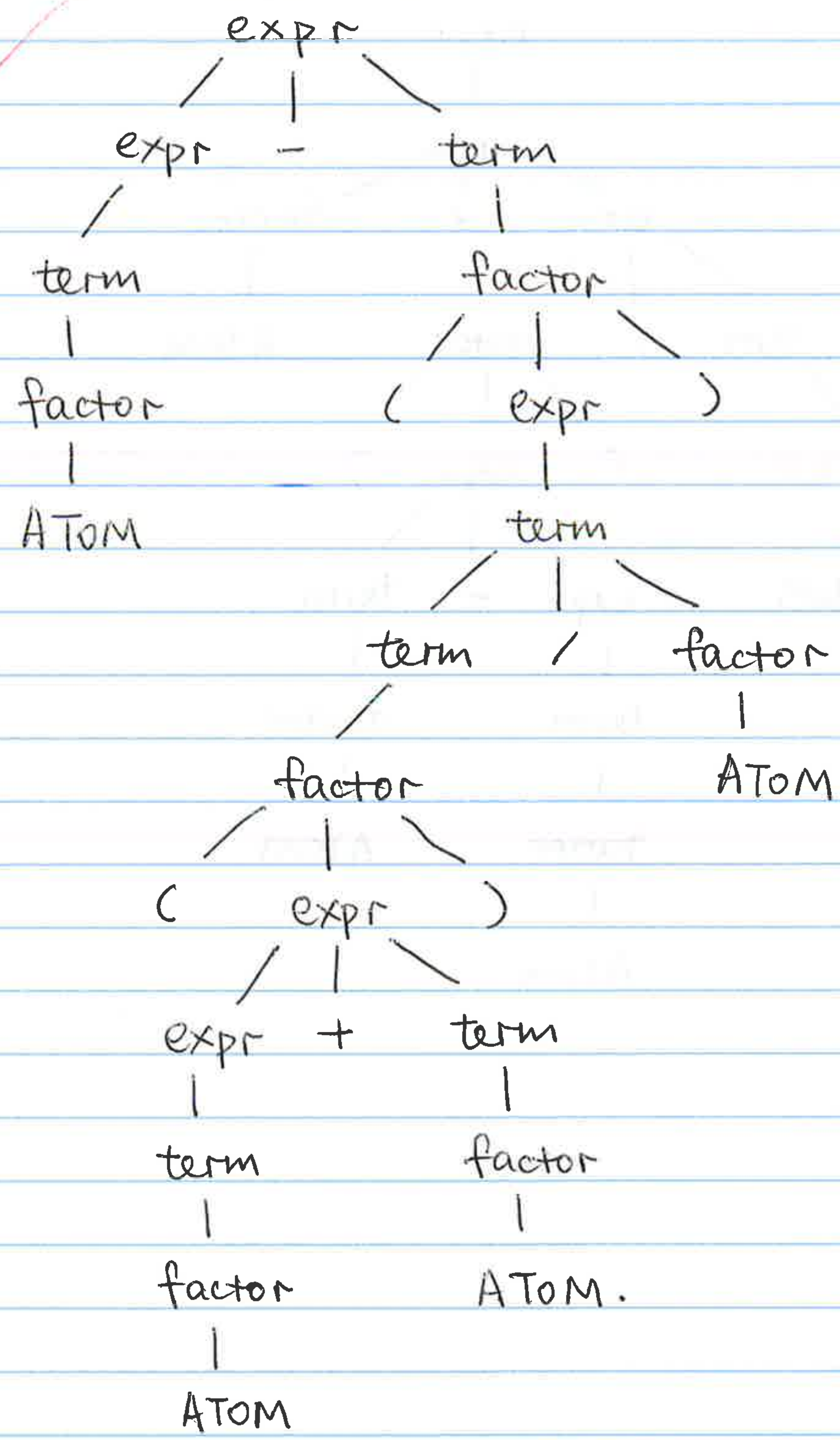
Cecilia Sui

3. ATOM / (ATOM − ATOM) * ATOM .



Cecilia Sui.

4. ATOM - ((ATOM + ATOM)/ ATOM)



CProg.

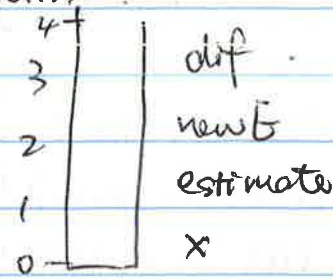
```
struct Node {
    int data
    struct Node* left;
    struct Node* right;
};
struct Node* insert (struct Node* root, int data) {
    if (root == NULL) {
        root = malloc (sizeof(struct Node));
        root -> data = data;
        root -> left = root -> right = NULL;
    }
    else if ( data < root->data) {
        root->left = insert (root -> left, data);
    }
    else if (data > root->data) {
        root -> right = insert ( root ->right, data);
    }
    return root;
}
...
int main() {
    struct Node * root ;    int entry ;
    scanf ("%d", & entry)
```

addr. of entry (ptr)

C → GSTAL.

```
NOP ; * Programmer — Bryan Crawley.
NOP ; * comments here.
NOP ; # include <stdio.h>.
```

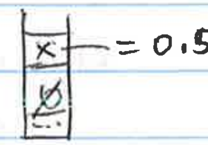
* Conceptual assign *.

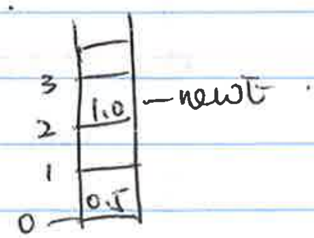


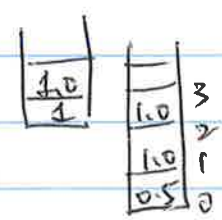
~~estim~~

~~x~~

```
ISP 4 ; (increment stack ptr).
LLI 58 ; Ascii for ":".
PTC.
LAA Ø
INF. ; Scanf("%f", &x).
STO ; datamem[Ø] = x.
LAA 2 ; addr. newE.
LLF 1.0 ; load 1.0.
STO. ; newE = 1.0.
LAA 1 ; est
LAA 2 ; newE.
LOD ; push datamem[2] => push 1.0.
STO ; datamem[1] = 1.0.
LAA 2 ; newE.
```







0.5 (est. x . est / +)*

```
LLF 0.5 ;
LAA 1 ; ~~x~~est.
LAA 0 ; x
LAA 1 ; est.

DUF ; x/est.
ADF ; ~~x/est~~ est + x/est.
MLF ; 0.5 * (est + x/est) ✓
```

print a char : LLI # ← ASCII
PTC

make room var : ISP # ← # of vars .

scanf("%f", &x) : ~~#~~ LAA # ← &X
INF
STO x= input .

scanf("%d", &x) : LAA # ← &x
INI .
STO x = input .

arithmetic :
eg. 0.5 * (est + x/est)
→ postfix 0.5 . (est . x . est . / +) * .

LLF 0.5
LAA # est .
~~LAA # x~~
LOD
LAA # x
LOD
LAA est # .
LOD .
DVF
ADF
MLF .

LAA 3 ; ~~dif*~~ &dif.
LAA 2 ; &newE.
~~LAA 1 ; est.~~
~~SBF~~ ;
~~LOD~~ ;
LAA 3; &dif.
LAA 2; &newE
LOD ;
LAA 1 ; est.
LOD ;
SBF ; newE - est
STO ; &dif = newE-est.

Ø - False
1 - True.

LAA 3; &dif.
LOD ; value of dif.
LLF -0.005;
LTF ; dif < -0.005 . 1 or Ø
LAA 3 ; &dif.
LOD ; dif.
LLF 0.005;
GTF ; dif > 0.005 1 or Ø.
ADI ; add Ø/1 together 0 1 2.

=Ø ~~✓~~ ×
≠Ø ~~×~~ ✓

LLI ~~1~~ Ø ; ~~1 or 2 > Ø ✓ Ø ≯ Ø ×.~~
~~GTI ; 1 or 2 > Ø ✓ Ø = Ø ×~~
~~GTI~~
1 ✓ Ø ×
EQI ; ~~if ≠ Ø~~ true if = Ø, false if ≠ Ø.
JPF loop while ; Jump if Ø (false).

Sui

10

```
//----------------------------------------------------------------------------------------
---------
// Author ------------- Cecilia Y. Sui
// Course ------------- Compiler Construction
// Instructor --------- Dr. Crawley
// Assignment --------- Binary Search Tree Implementation in C
// Data of Submission - August 27, 2019
//----------------------------------------------------------------------------------------
---------
```

maintain your data structure

```

#include <stddef.h>
#include<stdio.h>
#include<stdlib.h>

//----------------------------------------------------------------------------------------
---------
// Construction of Node
//----------------------------------------------------------------------------------------
---------
```

typedef ______ Name ;

```
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

//----------------------------------------------------------------------------------------
---------
// function to insert an element into the BST
//----------------------------------------------------------------------------------------
---------
```

it might not always.

always.

```
struct Node* insert(struct Node* root, int data){
    if (root == NULL){
        root = malloc(sizeof(struct Node));
```

→ check if ~~the~~ allocation succeeded

if (root == NULL) {
printf ("Insufficient memory");
exit(0);
}.

```
        root->data = data;
        root->left = root->right = NULL;
    }
    else if (data < root->data) {
        root->left = insert(root->left, data);
    }
    else if (data > root->data){
        root->right = insert(root->right, data);
    }
    return root;
};
```

← Unnecessary. Omit.

```

//----------------------------------------------------------------------------------------
---------
// inorder traversal of BST
//----------------------------------------------------------------------------------------
---------
void inorder(struct Node* root){
    if (root != NULL){
        inorder(root->left);
```

```
        printf("Content: %d \n", root->data);
        inorder(root->right);
    }
};
```

Unnecessary. Omit.

```

//----------------------------------------------------------------------------------------
---------
// main function
//----------------------------------------------------------------------------------------
---------
int main(){
    struct Node* root;
    int entry;
    root = NULL;
    printf("Entry: ");
    scanf("%d", &entry);
    while (entry != 0){
        root = insert(root, entry);
        printf("Entry: ");
        scanf("%d", &entry);
    }
    printf("\n");
    inorder(root);
    return 0;
};
```

```
[0] NOP; /*--------------------------------------------------------------
[1] NOP;  * Programmer--Cecilia Y. Sui
[2] NOP;  * Course------CS4223
[3] NOP;  * Project-----Homework #2
[4] NOP;  * Due---------September 10, 2019
[5] NOP;  *
[6] NOP;  * This program computes and displays an estimated square
[7] NOP;  * root.
[8] NOP;  *--------------------------------------------------------------
[9] ISP 4; make room for 4 variables
[10] NOP; /* Enter the raw data. */
[11] LLI 83; ASCII Code for 'S'
[12] PTC
[13] LLI 101; 'e'
[14] PTC
[15] LLI 108; 'l'
[16] PTC
[17] LLI 101; 'e'
[18] PTC
[19] LLI 99; 'c'
[20] PTC
[21] LLI 116; 't'
[22] PTC
[23] LLI 32; ' '
[24] PTC
[25] LLI 110; 'n'
[26] PTC
[27] LLI 117; 'u'
[28] PTC
[29] LLI 109; 'm'
[30] PTC
[31] LLI 98; 'b'
[32] PTC
[33] LLI 101; 'e'
[34] PTC
[35] LLI 114; 'r'
[36] PTC
[37] LLI 58; ':'
[38] PTC
[39] LLI 32; ' '
[40] PTC
[41] LAA 0; absolute address for float x
[42] INF
[43] STO
[44] NOP; /* An initial estimate of the square root */
[45] LAA 2; absolute address for float newEstimate
[46] LLF 1.0
[47] STO; newEstimate = 1.0
[48] NOP; /* Estimate the square root */
[49] LAA 2; float newEstimate
```

10

0 - x
1 - estimate
2 - newEstimate
3 - difference

We aren't ready for this yet.

Sui

```
[50] LAA 1; float estimate
[51] LAA 2; float newEstimate
[52] LOD
[53] STO
```

LAA 2 Now we're ready for it.

```
[54] LAA 0; float x
[55] LOD
[56] LAA 1; float estimate
[57] LOD
[58] DVF; calculate x/estimate
[59] LAA 1
[60] LOD
[61] ADF; calculate estimate + x/estimate
[62] LLF 0.5; load in float 0.5
[63] MLF; calculate 0.5*(estimate + x/estimate)
[64] STO; store value to newEstimate
```

Implement the expression directly from its postfix form.

Likewise for the Boolean expression below. Use addition to implement the Boolean OR.

```
[65] LAA 3; float difference
[66] LAA 2; float newEstimate
[67] LOD
[68] LAA 1; float estimate
[69] LOD
[70] SBF; calculate newEstimate - estimate
[71] STO; store value to difference
[72] LLF -0.005
[73] LAA 3
[74] LOD
[75] LEF; check if difference >= -0.005
[76] JPF 49; jump back to loop if difference < -0.005 (LEF -> False)
[77] LLF 0.005
[78] LAA 3
[79] LOD
[80] GEF; check if difference <= 0.005
[81] JPF 49; jump back to loop if difference > 0.005 (GEF -> False)
[82] NOP; /* Display the estimated square root */
[83] LLI 115; 's'
[84] PTC
[85] LLI 113; 'q'
[86] PTC
[87] LLI 114; 'r'
[88] PTC
[89] LLI 116; 't'
[90] PTC
[91] LLI 58; ':'
[92] PTC
[93] LLI 32; ' '
[94] PTC
[95] LAA 2; float newEstimate
[96] LOD
[97] PTF
[98] PTL
```

$7\frac{1}{2}$

Cecilia Y. Sui

Compiler Construction

Homework #3 Regular Expressions

September 17, 2019

1. Slic variable names:

    `[a-zA-Z][a-zA-Z0-9]*`

2. Slic reserved word while:

    `(w|W)(h|H)(i|I)(l|L)(e|E)`

w/o decimal pt.

eg. 6E23.

$-\frac{1}{2}$

3. Slic floating-point constants (allowing scientific notation e.g. 8.9e-3):

    `(-|+)?[0-9]+(.[0-9]+)?((e|E)((-|+)?)[0-9]+)?`

This allows numbers with no decimal point.

4. Slic character string constants:

    `"([^"]*("{2})*[^"]*)*"`

5. Over the alphabet {0, 1}, all non-negative binary integers that are divisible by four. Assume six-digit signed integers with two's complement.

    `0(0|1)(0|1)(0|1)00`

```
#--------------------------------------------------------------------
# Author ------- CeciliaY Y. Sui
# Course ------- Compiler Construction
# Assginment --- Project Checkpoint #1
# Description -- All Tokens in SLIC
#--------------------------------------------------------------------
# Regex                   # Name
+                         ADDITION
-                         SUBTRACTION
*                         MULTIPLICATION
/                         DIVISION
%                         MODULUS
<                         LESS
>                         GREATER
<=                        LESSEQUAL
>=                        GREATEREQUAL
=                         EQUAL
<>                        NOTEQUAL
&                         BOOLAND
|                         BOOLOR
~                         BOOLNOT
(                         LEFTPARENTH
)                         RIGHTPARENTH
[                         LEFTBRACKET
]                         RIGHTBRACKET
;                         SEMICOLON
:                         COLON
,                         COMMA
#                         COMMENT
:=                        ASSIGN
!                         CARRIAGERETURN
[mM][aA][iI][nN]                          MAIN
[eE][nN][dD] [mM][aA][iI][nN]             ENDMAIN
[eE][nN][dD]                              END
[eE][xX][iI][tT]                          EXIT
[iI][fF]                                  IF
[eE][lL][sS][eE]                          ELSE
[eE][nN][dD] [iI][fF]                     ENDIF
[wW][hH][iI][lL][eE]                      WHILE
[eE][nN][dD] [wW][hH][iI][lL][eE]         ENDWHILE
[tT][oO]                                  TO
[cC][oO][uU][nN][tT][iI][nN][gG]          COUNTING
[eE][nN][dD] [cC][oO][uU][nN][tT][iI][nN][gG]   ENDCOUNTING
[uU][pP][wW][aA][rR][dD]                  UPWARD
[dD][oO][wW][nN][wW][aA][rR][dD]          DOWNWARD
[rR][eE][aA][lL]                          REAL
[iI][nN][tT][eE][gG][eE][rR]              INTEGER
[dD][aA][tT][aA]                          DATA
[aA][lL][gG][oO][rR][iI][tT][hH][mM]      ALGORITHM
[rR][eE][aA][dD]                          READ
[pP][rR][iI][nN][tT]                      PRINT
[a-zA-Z][a-zA-Z0-9]*                      VARIABLENAME
"([^"]*("{2})*[^"]*)*"                    STRING
```

10

Quote this in Flex. Likewise for each below.

I'll be merciful - but the assignment said to double-space this document.

#.*

The entire comment should be a token, not just the "#" delimiter.

This should be two separate tokens: end and main.

`\"(([^"\n]*)|(\"\"))*\"`

[0-9]+ INTEGERREP

[-+]?[0-9]+~~*~~.[0-9]+~~*~~?([eE][-+]?[0-9]+)? REALREP [-+]? [0-9]+ . [0-9]+ ([eE][-+]?[0-9]+) ?

[\t\n\f...] WHITESPACE . ignore .

. TRASH .

(any other char, to catch it all).

printout to standard output .

Include —

- White space
- Trash

not in the grammar .

parsing fails .
(syntax error) .

E:"\n" — newline char.
→ for line count.

[-+]? [0-9]+((.[0-9]+ ([eE][-+]? [0-9]+)?) | (. [0-9]+))

integer : vector[25] ;

vector[i*2] := ~~~ ;

vector[3] := ~~ ;

whitespace . [\t\n\f\v] .

unary minus .

```
%{
/*
 * ==========================================================================
 * Author -------- Cecilia Y. Sui
 * Filename ------ Slice_scanner.l
 * Assignment ---- Checkpoint #2
 * Description --- Use flex to write a scanner for SLIC
 *
 * Version ------- This version prints out whitespace tokens
 * ==========================================================================
 */

#include <stdio.h>

%}

%%

#.*                            { printf("COMMENT:        %s\n", yytext); }
"+"                            { printf("ADDITION:       %s\n", yytext); }
"-"                            { printf("SUBTRACTION:    %s\n", yytext); }
"*"                            { printf("MULTIPLICATION: %s\n", yytext); }
"/"                            { printf("DIVISION:       %s\n", yytext); }
"%"                            { printf("MODULUS:        %s\n", yytext); }
"<"                            { printf("LESSTHAN:       %s\n", yytext); }
">"                            { printf("GREATERTHAN:    %s\n", yytext); }
"<="                           { printf("LESSOREQUAL:    %s\n", yytext); }
">="                           { printf("GREATEROREQUAL: %s\n", yytext); }
"="                            { printf("EQUAL:          %s\n", yytext); }
"<>"                           { printf("NOTEQUAL:       %s\n", yytext); }
"&"                            { printf("BOOLAND:        %s\n", yytext); }
"|"                            { printf("BOOLOR:         %s\n", yytext); }
"~"                            { printf("BOOLNOT:        %s\n", yytext); }
"("                            { printf("LEFTPARENTH:    %s\n", yytext); }
")"                            { printf("RIGHTPARENTH:   %s\n", yytext); }
"["                            { printf("LEFTBRACKET:    %s\n", yytext); }
"]"                            { printf("RIGHTBRACKET:   %s\n", yytext); }
";"                            { printf("SEMICOLON:      %s\n", yytext); }
":"                            { printf("COLON:          %s\n", yytext); }
```

10

```
","                           { printf("COMMA:          %s\n", yytext); }
":="                          { printf("ASSIGN:         %s\n", yytext); }
"!"                           { printf("CARRIAGERETURN: %s\n", yytext); }
[mM][aA][iI][nN]                                      { printf("MAIN:           %s\n", yytext); }
[eE][nN][dD][ ][mM][aA][iI][nN]                       { printf("ENDMAIN:        %s\n", yytext); }
[eE][xX][iI][tT]                                      { printf("EXIT:           %s\n", yytext); }
[iI][fF]                                              { printf("IF:             %s\n", yytext); }
[eE][lL][sS][eE]                                      { printf("ELSE:           %s\n", yytext); }
[eE][nN][dD][ ][iI][fF]                               { printf("ENDIF:          %s\n", yytext); }
[wW][hH][iI][lL][eE]                                  { printf("WHILE:          %s\n", yytext); }
[eE][nN][dD][ ][wW][hH][iI][lL][eE]                   { printf("ENDWHILE:       %s\n", yytext); }
[tT][oO]                                              { printf("TO:             %s\n", yytext); }
[cC][oO][uU][nN][tT][iI][nN][gG]                      { printf("COUNTING:       %s\n", yytext); }
[eE][nN][dD][ ][cC][oO][uU][nN][tT][iI][nN][gG]       { printf("ENDCOUNTING:    %s\n", yytext); }
[uU][pP][wW][aA][rR][dD]                              { printf("UPWARD:         %s\n", yytext); }
[dD][oO][wW][nN][wW][aA][rR][dD]                      { printf("DOWNWARD:       %s\n", yytext); }
[rR][eE][aA][lL]                                      { printf("REAL:           %s\n", yytext); }
[iI][nN][tT][eE][gG][eE][rR]                          { printf("INTEGER:        %s\n", yytext); }
[dD][aA][tT][aA]                                      { printf("DATA:           %s\n", yytext); }
[aA][lL][gG][oO][rR][iI][tT][hH][mM]                  { printf("ALGORITHM:      %s\n", yytext); }
[rR][eE][aA][dD]                                      { printf("READ:           %s\n", yytext); }
[pP][rR][iI][nN][tT]                                  { printf("PRINT:          %s\n", yytext); }
[a-zA-Z][a-zA-Z0-9]*                                  { printf("VARIABLENAME:   %s\n", yytext); }
\"(([^"\n"]*)|(\"\"))*\"                              { printf("STRING:         %s\n", yytext); }
[0-9]+                                                { printf("INTEGERREP:     %s\n", yytext); }
[-+]?[0-9]+(("."[0-9]+)|(("."[0-9]+)?[eE][-+]?[0-9]+)) { printf("REALREP:        %s\n", yytext); }
[ \t\n\r\f\v]                                         { /* printf("WHITESPACE:   %s\n", yytext);*/ }
.                                                     { printf("TRASH:          %s\n", yytext); }

%%

/* Main Program to call yylex() */
int main()
{
    yylex();
    return 0;
}
```

{if(DEBUG) printf(~);}

C macros

macro instructions.

#ifndef DEBUG

:

#endif.

#define DEBUG 0 or 1. manifest constant.

preprocessor instruction.

turn off

turn on.

10

Good job

I am only interested in the grammar on this assignment. I'm ignoring the rest of it.

```
%{
/*
 * ==============================================================================
 *
 * Author ---------------- Cecilia Y. Sui
 * Course ---------------- Compiler Construction
 * Date of Submission ---- October 7, 2019
 *
 * Assignment ------------ Checkpoint #3 SLIC Context-Free Grammar in Bison Format
 *
 * ==============================================================================
 */

#include <stdio.h>

%}

/* C union declaration */
%union {
   int intval;
   float realval;
   char *sval;
}
```

→ pass data from scanner to parser.

string value.

```
/* Declare Tokens */
%token ADDITION SUBTRACTION MULTIPLICATION DIVISION MODULUS
%token LESSTHAN GREATERTHAN LESSOREQUAL GREATEROREQUAL EQUAL NOTEQUAL
%token AND OR NOT
```

move to one per line.

Cecilia Sui

1 tab indentation .

```
%token LPARENTH RPARENTH
%token LBRACKET RBRACKET
%token SEMICOLON
%token COLON
%token COMMA
%token ASSIGN
%token CARRIAGERETURN
%token MAIN
%token END
%token EXIT
%token IF ELSE
%token WHILE
%token TO
%token COUNTING UPWARD DOWNWARD
%token REAL INTEGER
%token DATA ALGORITHM
%token READ PRINT
%token <sval>      VARIABLE
%token <sval>      STRING
%token <intval>    INTCONSTANT
%token <realval>   REALCONSTANT
%token NEWLINE
%token WHITESPACE
%token TRASH

%%
```

string value .

Cecilia Sui

```
/* -------------------------------------
 Context Free Grammar Rules --------------------------------------- */

/* ------------------------ Main Program ------------------------ */

program         : MAIN SEMICOLON data algorithm END MAIN SEMICOLON {}
                ;

/* ------------------------ Data Section ------------------------ */

data            : DATA COLON fulldeclaration {}
                ;

fulldeclaration : declaration{}
                | declaration fulldeclaration {}
                ;

declaration     : REAL COLON varlist SEMICOLON {}
                | INTEGER COLON varlist SEMICOLON {}
                ;

varlist         : arrayvar {}
                | arrayvar COMMA varlist {}
                ;

arrayvar        : VARIABLE {}
                | VARIABLE LBRACKET INTCONSTANT RBRACKET {}
                ;
```

Omit. Likewise below.

Cecilia Sui

```
/* --------------------------- Algorithm Section --------------------------- */

algorithm       : ALGORITHM COLON body {}
                ;

body            : /* empty */ {}
                | statement body {}
                ;

statement       : assignment {}
                | counting {}
                | ifstatement {}
                | whileloop {}
                | read {}
                | print {}
                | exit {}
                ;

assignment      : variable ASSIGN fullexpression SEMICOLON {}
                ;

exit            : EXIT SEMICOLON {}
                ;

/* ---------------------------- control structures ------------------------------- */
```

ifelsestmt

Cecilia Sui

separate up & down

```
counting        : COUNTING variable UPWARD fullexpression TO fullexpression SEMICOLON body
 END COUNTING {}
                |
 COUNTING variable DOWNWARD fullexpression TO fullexpression SEMICOLON body END COUNTING
{}
                ;
```

separate to 2 if & ifelse stmt

```
ifstatement     : IF fullexpression SEMICOLON body END IF SEMICOLON {}
                | IF fullexpression SEMICOLON body ELSE SEMICOLON body END IF SEMICOLON {}
                ;

whileloop       : WHILE fullexpression SEMICOLON body END WHILE SEMICOLON {}
                ;

/* ----------------------- variable --> fullexpression ----------------------- */

fullexpression : comparison {}
                | NOT fullexpression {}
                | fullexpression AND comparison {}
                | fullexpression OR comparison {}
                ;

comparison      : expression {}
                | comparison LESSTHAN expression {}
                | comparison GREATERTHAN expression {}
                | comparison LESSOREQUAL expression {}
                | comparison GREATEROREQUAL expression {}
                | comparison EQUAL expression {}
```

Cecilia Sui

```
                | comparison NOTEQUAL expression {}
                ;

expression      : term {}
                | expression ADDITION term {}
                | expression SUBTRACTION term {}
                ;

term            : factor {}
                | term MULTIPLICATION factor {}
                | term DIVISION factor {}
                | term MODULUS factor {}
                ;

factor          : atom {}
                | LPARENTH expression RPARENTH {}
                | SUBTRACTION factor { /* used for unary minus */ }
                ;

atom            : variable {}
                | INTCONSTANT {}
                | REALCONSTANT {}
                ;

variable        : VARIABLE {}
                | VARIABLE LBRACKET expression RBRACKET {}
                ;
```

Cecilia Sui,

```
/* ----------------------- read & print ----------------------- */

read            : READ variable SEMICOLON {}
                ;

print           : PRINT printlist SEMICOLON {}
                ;

printlist       : printitem {}
                | printitem COMMA printlist {}
                ;

printitem       : fullexpression {}
                | STRING {}
                | CARRIAGERETURN {}
                ;

assignment      : VARIABLE ASSIGN fullexpression {}
                ;

%%
```

10

a.exe
-lfl } → for turn in.

(clean it b/f).
Turn in only source code.
Don't turn in test cases!

```
%{

/*
 *
==========================================================================
 * Author ---------------- Cecilia Y. Sui
 * Course ---------------- Compiler Construction
 * Filename -------------- Slic_parser.y
 * Date of Submission ---- October 20, 2019
 * Assignment ------------ Checkpoint #4
 * Description ----------- bison parser for SLIC data section
 *
==========================================================================
 */

#include <stdio.h>
#include <stdlib.h>

int yyerror();
int yylex();

%}

/* C union declaration */
%union {
   int intval;
   float realval;
   char *sval;
}

/* Declare Tokens */
%token ADDITION
%token SUBTRACTION
%token MULTIPLICATION
%token DIVISION
%token MODULUS
%token LESSTHAN
%token GREATERTHAN
%token LESSOREQUAL
%token GREATEROREQUAL
%token EQUAL
%token NOTEQUAL
%token AND
%token OR
%token NOT
%token LPARENTH
%token RPARENTH
%token LBRACKET
%token RBRACKET
%token SEMICOLON
%token COLON
%token COMMA
%token ASSIGN
%token CARRIAGERETURN
```

Good job. My (few) test cases revealed no defects.

Turn off DEBUG (==0)

Naming files:
scanner.l
parser.y

Distinguish by 1st char for tab completion.

newline char \n.
eg. print a, "howdy", ! → carriage return.

newline token → end of line

\n. → line cnt. char cnt. word cnt. → flex has an automatic way to do it.

[\n]

"\n"|"\r\n" for newline EOL.

yyerror().

```
%token MAIN
%token END
%token EXIT
%token IF
%token ELSE
%token WHILE
%token TO
%token COUNTING
%token UPWARD
%token DOWNWARD
%token REAL
%token INTEGER
%token DATA
%token ALGORITHM
%token READ
%token PRINT
%token <sval>     VARIABLE
%token <sval>     STRING
%token <intval>   INTCONSTANT
%token <realval>  REALCONSTANT
%token NEWLINE
%token TRASH

%%

/* ------------------------------------- Grammar Rules ------------------
------------------- */

/* ------------------------ Main Program ------------------------ */

program         : MAIN SEMICOLON data algorithm END MAIN SEMICOLON
                ;

/* ------------------------ Data Section ------------------------ */

data            : DATA COLON fulldeclaration
                ;

fulldeclaration : declaration{}
                |  declaration fulldeclaration
                | /* empty to allow empty declaration */
                ;

declaration     : datatype COLON varlist SEMICOLON

datatype        : REAL
                | INTEGER
                ;

varlist         : item
                | item COMMA varlist
                | /* empty to allow empty varlist */
                ;
```

```
item            : VARIABLE
                | VARIABLE LBRACKET INTCONSTANT RBRACKET
                ;

/* ----------------------- Algorithm Section ----------------------- */

algorithm       : ALGORITHM COLON body
                ;

body            : /* empty to allow empty body */
                ;


%%

int yyerror() {
   printf("Called yyerror()\n");
   return  0;
}
```

Name: Cecilia Sui

Assignment: Checkpoint #5

Course: CS4223 Semester: Fall 2019

Score: 10

Remarks:

- You're getting a reduce/reduce conflict warning from Bison. See me for assistance in eliminating it.
- When you use code numbers, I recommend you use symbolic names for them (an enum or a #define manifest constant).
- You're doing well. Keep on going!

code generator → output to array → backpatch for jump instr.

→ then output contents in array.

dynamically allocated array → grow as needed.

Checkpt #8 submission b/f. Tues.

Name: Cecilia Sui

Assignment: Checkpoint #6 / Exam #2

Course: CS4223 Semester: Fall 2019

Score: 95

Remarks:

✓ resolved.

- Type coercions are not working correctly. For example, where x is real, this assignment:

  x := 12 * 0.5;

  should coerce the 12 to be a float; but instead it does two ITF instructions after loading the 0.5 onto the stack.

✓ fixed.

- You should write only GSTAL code to the screen. If you write anything else, I am unable to capture and execute the GSTAL code without editing it by hand to delete the other stuff.

reverse orders.

↓

recurse left & right in post order.

- For the Boolean operators, you evaluate the operands in reverse order. I don't suppose it hurts anything to do so, but it sets a risky precedent. If we had functions with side effects, it would be more problematic.

print. if. ifelse. read. exit. while. counting loops.

GSTAL

The Georgetown Stack Assembly Language

Bryan Crawley

$\boxed{\text{HLT}}$ ☺ . or ↵ for new line

Introduction

The Georgetown Stack Assembly Language (GSTAL) is derived from STAL, a stack assembly language designed by Gerald Wildenberg of St. John Fisher College, Rochester, NY. [1]

args are allowed ✓

The GSTAL virtual machine is a *zero-address machine*. The machine instructions do not include memory addresses. They retrieve their operands from a central stack, and they push their results onto the same stack. The machine's memory architecture comprises code memory, data memory, and three special-purpose registers.

Harvard Architecture : code & data memory are separate .
eg. microcontrollers .

Code Memory

ISP to make space to store variables .

Each code-memory location holds one GSTAL instruction. The first instruction of the current GSTAL program resides at address zero, with addresses increasing consecutively for subsequent instructions. The size of code memory is limited only by the size of the process in which the GSTAL interpreter runs. That is, there is no inherent upper bound on code addresses.

Data Memory

32-bit .

Data memory is an array of 4-byte words, addressable by word and not by byte. The first word resides at address zero, with addresses increasing consecutively for subsequent words. Each memory word can contain either an integer or a floating-point number. GSTAL instructions treat data memory as a stack. The top-of-stack pointer is in the special register called `tos`. (See Special Registers below.) Each instruction fetches its operands by popping them from the stack. The result of the operation, if any, is pushed back onto the stack. When a GSTAL program begins running, the stack is initially empty.

Some of the GSTAL instructions violate the stack abstraction by addressing data memory words other than the top of the stack. These instructions make it possible to store and retrieve variables. See the following pages for complete descriptions of the GSTAL instructions.

The size of data memory is limited only by the size of the process in which the GSTAL interpreter runs. That is, there is no inherent upper bound on data addresses. However, any address reference less than zero or greater than `tos` (see Special Registers below) is erroneous and results in a run-time error.

backpatching .

Special Registers

The GSTAL virtual machine has three special registers called `tos`, `pc`, and `act`. The registers cannot be addressed directly. Rather, their values are altered as side effects of the various GSTAL instructions. The operational semantics on the following pages describe how the registers are manipulated by each instruction. A description of each register follows.

`tos` The address in data memory of the current top entry of the stack. Any address reference greater than `tos` or less than zero is invalid and will result in an execution error. When the stack is empty, `tos` is undefined.

`pc` The program counter. This is the address in code memory of the current GSTAL instruction. The initial value is zero.

`act` The base address in data memory of the current activation frame. This register is relevant only to subroutine calls, returns, and parameter passing.

Input and Output

All GSTAL input comes from the standard input. All output goes to the standard output. The input instructions can read integers and floating-point numbers. The output instructions can write integers, floating-point numbers in exponential form, and individual characters.

Comments and Blank Lines

You can append a comment to the right-hand side of any GSTAL instruction. A comment consists of a semicolon (;) followed by any text extending to the right-hand end of the line. GSTAL does not permit blank lines or lines that contain only a comment with no instruction. Every line of a GSTAL program must contain a GSTAL instruction.

Every line must have an instruction. ☆

The GSTAL Interpreter

The GSTAL interpreter runs any valid GSTAL program. Before it executes the GSTAL code, it scans the entire program to verify the syntax. If it finds any syntax errors, it reports the errors and aborts the run. If it finds no syntax errors then it runs the program. The GSTAL program terminates under any of these three conditions:

- A `HLT` instruction is executed.
- The physically last statement of the program is executed, and it is neither a `JMP`, `JPF`, nor `RET` that transfers control to another place in the program. In other words, the program halts if it "falls through the bottom" without executing a `HLT` instruction.
- An execution error occurs in the GSTAL code. The interpreter reports all execution errors with appropriate error messages.

$ cd gstal

$make . $ ./gstal calendar.gstal .

* prog. must end at a newline character * .

Use this syntax to run a GSTAL program at the command-line prompt:

```
gstal <filename>
```

where `<filename>` is the name of a text file that contains a GSTAL program. For example, if you have a GSTAL program in a file called `proj1.g`, then do this:

```
gstal proj1.g
```

Interpreter Options

The interpreter includes two options that are helpful in debugging GSTAL programs. The `-d` option runs the program and produces a stack dump if an execution error occurs. The stack dump is written to a text file called `stackdump`. For example:

```
gstal -d proj1.g
```

The `-l` (lowercase "L") option does not run the program, but instead writes a numbered listing of the program to the standard output. This helps you identify line numbers which may be the targets of `JMP`, `JPF`, or `CAL` instructions. For example:

```
gstal -l proj1.g
```

If you want to save the numbered listing in a file, then redirect the standard output to a text file of your choosing. For example, to write the numbered listing to a file called `proj1.listing`, do this:

```
gstal -l proj1.g > proj1.listing
```

References

[1] 1990. Wildenberg, Gerald. *Using a Stack Assembler Language in a Compiler Course*, *SIGCSE Bulletin* **22**, *No. 4*: p. 43 (December).

Integer Arithmetic

Op Code	Description	Semantics	Argument

ADI	Addition	b = pop();
		a = pop();
		push(a+b);	

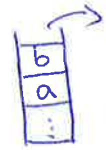
SBI	Subtraction	b = pop();
		a = pop();
		push(a-b);	

MLI	Multiplication	b = pop();
		a = pop();
		push(a*b);	

DVI	Division	b = pop();
		a = pop();
		push(a/b);	

NGI	Negation	a = pop(); push(-a);	

add 2 # on top of stack.



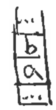
a - b → postfix ab -

popped 2nd

popped 1st

2nd/1st

a / b → a b /



Floating-Point Arithmetic

Op Code	Description	Semantics	Argument

ADF	Addition	y = pop();
		x = pop();
		push(x+y);	

SBF	Subtraction	y = pop();
		x = pop();
		push(x-y);	

MLF	Multiplication	y = pop();
		x = pop();
		push(x*y);	

DVF	Division	y = pop();
		x = pop();
		push(x/y);	

NGF	Negation	x = pop(); push(-x);	

Integer Relational Operations

Op Code	Description	Semantics	Argument

EQI	Equal To	b = pop();
		a = pop();
		push(a==b);
			zero if false.
			nonzero → True.
			(not always 1).

NEI	Not Equal To	b = pop();
		a = pop();
		push(a!=b);	

LTI	Less Than	b = pop();
		a = pop();
		push(a<b);	

LEI	Less Than Or Equal To	b = pop();
		a = pop();
		push(a<=b);
			a <= b

GTI	Greater Than	b = pop();
		a = pop();
		push(a>b);	

GEI	Greater Than Or Equal To	b = pop();
		a = pop();
		push(a>=b);	

Floating-Point Relational Operations

Op Code	Description	Semantics	Argument

EQF	Equal To	y = pop();
		x = pop();
		push(x==y);	

NEF	Not Equal To	y = pop();
		x = pop();
		push(x!=y);	

LTF	Less Than	y = pop();
		x = pop();
		push(x<y);	

LEF	Less Than Or Equal To	y = pop();
		x = pop();
		push(x<=y);	

GTF	Greater Than	y = pop();
		x = pop();
		push(x>y);	

GEF	Greater Than Or Equal To	y = pop();
		x = pop();
		push(x>=y);	

Data Type Conversion

Op Code	Description	Semantics	Argument
FTI	Floating-Point to Integer truncate ? round?	x = pop(); push((int) x);	bit manipulation, change representation.
ITF	Integer to Floating-Point	a = pop(); push((float) a);	Int. truncation loose fractional part.

Input and Output

Op Code	Description	Semantics	Argument
PTI	Print Integer	a = pop(); printf("%d",a);	
PTF	Print Floating-Point	x = pop(); printf("%e",x); → Scientific representation.	
PTC	Print Character	a = pop(); printf("%c",a); 1 char at a time use ASCII #	
PTL	Print Newline Character	printf("");	
INI	Input Integer → user input.	scanf("%d", &a); push(a);	
INF	Input Floating-Point	scanf("%f", &x); push(x);	

Stack Manipulation

Op Code	Description	Semantics	Argument
LLI <arg>	Load Literal Integer	push(arg);	<arg> is an integer.
LLF <arg>	Load Literal Floating-Point	push(arg);	<arg> is a floating-point number.
ISP <arg>	to make room for variables. Increment Stack Pointer	tos = tos + arg;	<arg> is a non-negative integer.
DSP <arg>	Decrement Stack Pointer	tos = tos - arg;	<arg> is a non-negative integer.
STO	Store	b = pop(); → value a = pop(); → memory addr. datamem[a] = b;	
STM	Store Memory	b = pop(); → memory addr. a = pop(); → value. datamem[b] ='a; push(b); → push on top of stack the value b.	
LOD	Load	a = pop(); → mem. addr. push(datamem[a]); → push value at mem addr.[a] on top of stack.	

Flow Control

Op Code	Description	Semantics	Argument
LAA <arg>	Load Absolute Address push the mem. addr. to stack. (then access value thru LLI to write in value).	push(arg);	<arg> is a non-negative integer.
? LRA <arg>	Load Relative Address	push(act+arg);	<arg> is a non-negative integer.
JMP <arg>	Unconditional Jump to a mem addr.	pc = arg;	<arg> is a non-negative integer.

JPF <arg>	Jump If False	a = pop();
		if (a==0) — False.
		pc = arg;
			<arg> is a non-negative integer;

PAR <arg>	Load Parameter Address	push(act-arg);	<arg> is a non-negative integer.

CAL <arg>	Call Subroutine	push(act);
		act = tos;
		push(pc);
		pc = arg;
			<arg> is a non-negative integer.

RET	Return From Subroutine	pc = datamem[act+1] + 1;
		tos = act-1;
		act = datamem[act];	

NOP	No Operation	<do nothing>	
HLT	Halt	<execution terminates>	

Name: Cecilia Y. Sui

Examination #1

CS 4223-01 — October 3, 2019

Short Answer (20 points) Total Score: 96½

1. Briefly discuss the three registers of the GSTAL virtual machine.

tos – top of stack register ; points to the top of the stack. It's undefined when stack is empty.

pc – program counter ; points to the current instruction. It's at zero at the start of a program.

act – register for the base address of the current activation frame

2. Consider the typical compilation process, as in a production-quality compiler. Identify the three parts of the *synthesis phase*.

synthesis phase (backend)

(1) machine-independent code optimizer
↓
(2) code generator
↓
(3) machine-dependent code optimizer.
⟶ (Target Code).

3. Explain what a *lexeme* is.

a lexeme is what character(s) that matched the current regular expression. When yylex() is called, the flex scanner scans the input string, and attempts to find a match that matches a regular expression in the second section of the flex program in a top ~~to bottom~~ priority. When a match is found, the string matched to the regular expression is the lexeme. * the string from the input that matches a token.

4. State Chomsky's hierarchy of languages. It is sufficient to name the layers of the hierarchy and arrange them in their correct hierarchy order.

Type 0 Recursively Enumerable Languages.
1 Context-sensitive Languages
2 Context-Free Languages
3 Regular Languages.

20

Multiple Choice (20 points)

Mark the one best answer for each of these multiple-choice questions.

D 1. What does it mean to say a computer has *von Neumann architecture*?

a. The lowest address is 0.
b. Code and data are stored in ~~separate~~ same memory units.
c. GSTAL instructions, in most cases, do not specify the addresses of operands.
d. Code and data are stored in the same memory unit.
e. None of these

E 2. The products produced by a parser are a symbol table and a(n):

a. token stream
b. parse tree
c. context-free grammar
d. regular expression
e. abstract syntax tree

B 3. Regular expressions have three fundamental operations. What are they?

a. sequence, selection, and repetition
b. concatenation, alternation, and Kleene closure
c. positive closure, Kleene closure, and negative closure
d. concatenation, substring, and positive closure
e. and, or, and complement

B 4. Which of these is the pointer dereferencing operator in C?

a. & b. * c. | d. ? e. ~

E 5. Name the C library function that allocates a given number of bytes in the heap.

a. *new()*
b. *allocate()*
c. *halloc()*
d. *reserve()*
e. *malloc()*

10

set of strings.

B 6. Consider our formal definition of *language*. All languages are infinite. / or finite.

a. True
b. False

D 7. A *flex* input file is divided into three sections. The third is the ___________ section.

a. rules — regex ②
b. data
c. definitions — ①
d. functions — C code ③.
e. productions ×

① definition
② rules
③ user subroutine / c code

B 8. The code generator is part of the ______________ of a compiler.

a. analysis phase (front end).
b. synthesis phase (backend)
c. front end
d. parser
e. syntax analyzer

Source → Lexical → Syntax → Semantic → Intermediate Code gen. ⇓ . . . . .

⇓ machine-independent code optimizer → code generator → machine-dependent code optimizer → Target Prog.

A 9. Which of these will generate a scanner for us?

a. *flex* — scanner
b. *bison* — parser

A 10. Are you enjoying the exam so far?

a. Yes
b. No
c. I'd rather not say.

It's an old favorite.

10

Flex (20 points)

Consider a simple language that has these tokens:

- Variable names
- Floating-point constants with no exponent; only digits and a decimal point.
- The reserved word "bisons", which is not case sensitive
- White space (tabs and spaces)

Create a *flex* scanner for this language. In the C semantic code for each token, write the lexeme to the screen, and do not return. In the *user functions* section of the *flex* code, write a small main program to call the scanner.

```
%{
#include <stdio.h>
%}
%%
[a-zA-Z][a-zA-Z0-9]          { printf("%s\n", yytext); }
[0-9]+"."[0-9]+              { printf("%s \n", yytext); }
[bB][iI][sS][oO][nN][sS]     { printf("%s \n", yytext); }
[ \t]+                       { printf("%s\n", yytext); }

%%
int main() {
    yylex();
    return 0;
}
```

ptr. to a null-terminated array of characters

variable provided by yylex().

must return an integer.

Flex
%{
#include .
int a = 0;
%}
%%.
[—] { ~ }.
%%
int main()

Bison
%{
#include< stdio.h>
int yyerror();
%}
% token
% type
%%.
<CFG>
%% main {}.

20

Grammars (10 points)

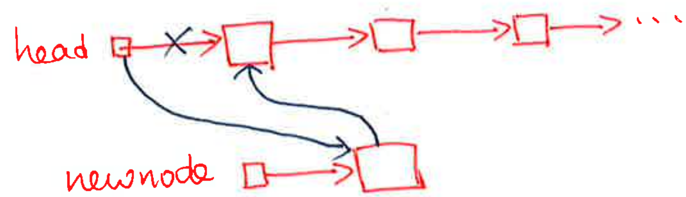
Write a context-free grammar that defines the syntax of arithmetic expressions. Include the four binary operators (+, -, *, /), parentheses, and variables represented by the token VARIABLE. Your grammar should enforce the standard priority and associativity of the operators.

```
expression  -->  expression + term
expression  -->  expression - term
expression  -->  term
term        -->  term * factor
term        -->  term / factor
term        -->  factor
factor      -->  ( expression )
factor      -->  VARIABLE
```

Stuff About C (10 points)

1. Complete the C program fragment below to declare a *struct* that can serve as a node of a singly-linked list. Let the node contain one integer and one floating-point number.

```
struct listnode {
        int data;
        float data2;
        struct listnode* node;
};
```



-1½

2. Assume that *head* is a C pointer variable that points to the head node of a list formed from the nodes declared above. Write a program fragment that allocates a new node, places an integer and a floating-point number into the relevant fields (you choose the values), and links the node into the head of the list. Declare any variables that you use in addition to *head*.

```
struct listnode* newnode;
        <--- newnode = malloc(sizeof(struct listnode));
newnode -> data = 5;
newnode -> data2 = 3.14;
newnode -> node = head;
head = newnode;
```

newnode -> data }
(*newnode).data } -> equivalent.

L repoint the head to the newnode created.

18½

GSTAL (20 points)

Translate this C program to GSTAL. Number the lines of your GSTAL code so I can see where the jumps are going. Include a comment to document the addresses of the variables.

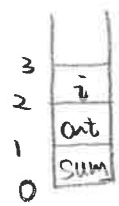
```
#include <stdio.h>

int main()
{
   int sum, count, i;

   sum = 0;
   for (count = 1; count <= 100; count++) {
      sum = sum + count;
   }

   printf("%d\n", sum);

   return 0;
}
```



count = 1
conditional jump → while (count <= 100) {
sum = sum + count;
count = count + 1 ;
unconditional jump. → }.

top-test loop ✱ .
(2 jumps) .

Line #

0 NOP ; addr : sum 0, count 1, i 2 .
1 ISP 3
2 LAA 0
3 LLI 0
4 STO ; sum = 0
5 LAA 1
6 LLI 1
7 STO ; count = 1
8 LAA 0
9 LAA 0
10 LOD
11 LAA 1
12 LOD
13 ADI
14 STO ; sum = sum + count .
15 LAA 1
16 LAA 1
17 LOD
18. LLI 1
19. ADI
20. STO ; count = count + 1 .
21. LAA 1
22. LOD
23. LLI 100
24 GTI ; count > 100
25. JPF 8
26. LAA 0
27. LOD
28. PTI
29. PTL
30. HLT

This implements a bottom-test loop that is not equivalent to the top-test for-loop.

−2

18