Programming Assignment #1.
Flex & Bison Exercise

Prof. Jaeseung Choi

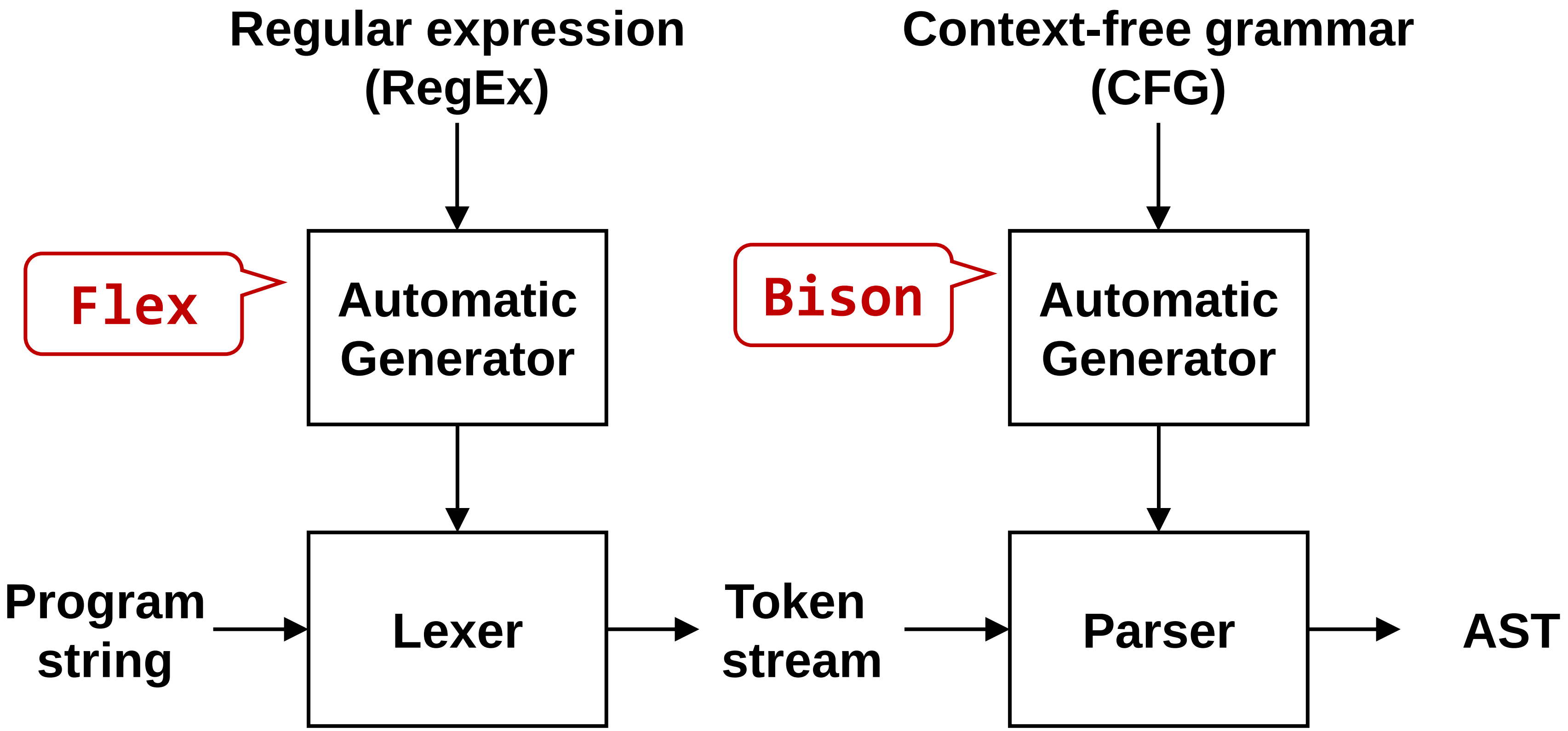
Dept. of Computer Science and Engineering

Sogang University

General Information

- **Check the "Programming Assignment #1" post in the *Assignment* tab of *Cyber Campus***
  - Example (**`Ex.tgz`**) and skeleton code (**`HW1.tgz`**) are attached
  - Deadline: **10/16 Wednesday 23:59**
  - Submission will be accepted in that post, too
  - Late submission deadline: **10/18 Friday 23:59** **(-20% penalty)**
  - **CSPRO server is experiencing a trouble recently**
  - **If the issue persists, I will extend the deadline**
- **Please read the instructions in this slide carefully**
  - This slide is a step-by-step tutorial for `Flex` and `Bison`
  - It also contains important submission guidelines
    - If you do not follow the guidelines, you will get penalty

Automatic Front-End Generation



`Flex/Bison` vs. `Lex/Yacc`?

- **`Lex/Yacc` are more popular names than `Flex/Bison`**
  - In fact, **`Lex/Yacc`** are the names of old tools
  - **`Flex`** (Fast Lex) is rewritten version of **`Lex`** (with better performance and more permissive license)
  - Similarly, **`Bison`** is rewritten version of **`Yacc`**
- **In most Linux systems, `lex` and `yacc` commands are automatically redirected to `flex` and `bison`**
  - Still, many people just call them **`Lex`** and **`Yacc`**, even when they are actually using **`Flex`** and **`Bison`**

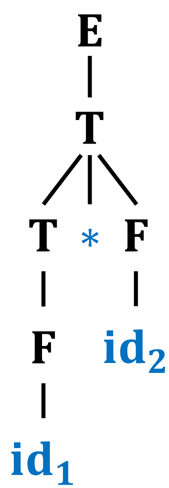
`Bison`: LALR Parsing

- **`Bison` generates an LALR parser for the provided CFG**
  - **LALR** is one kind of *bottom-up* parsing
- **Although we will not learn the details of LALR parsing in this course, you can still use `Bison`**
  - All you have to know is how to deal with CFG
  - Also, it shares similar principles with **SLR parsing** that you will learn in **Chapter 4. Bottom-up Parsing**

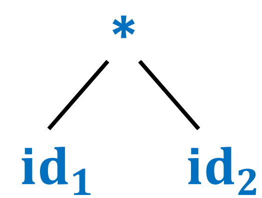
Goal of This Assignment

- **The first goal is learning how to use `Flex` and `Bison` to generate front-end code**
- **Another important topic here is construction of AST**
  - In **Chapter 3** and **4**, we focused on **parse tree**
  - However, **AST** is more concise and desirable form for compiler
  - Our `Bison`-generated parser will construct AST *on the fly*

Parse tree of $\mathbf{id}_1 * \mathbf{id}_2$



Abstract syntax tree (AST) form



I will first explain the basics of `flex` and `bison` with a simple example

Then, we will move on to the actual skeleton code for the assignment

Structure of `.l` and `.y` File

- **Both `lex(flex)` file (`.l`) and `yacc(bison)` file (`.y`) have the following structure**

```
%{
```

Inclusion of C header files and declaration of C variables (this part is already done for you)

```
%}
```

Declaration (e.g., tokens, start symbol, …)

```
%%
```

Definition (e.g., production rules, …)

```
%%
```

If needed, more functions like **`main()`** can be defined here (this part is already done for you)

Getting Started: Example CFG

- **Assume the following CFG for numeric expressions**
  - Recall that we used this CFG as an example in the lecture
  - Also, recall that **num** token can actually represent various integers, so it must be associated with a concrete integer value
  - Similarly, **id** token must be associated with a concrete string
  - So, how should we provide this CFG to the parser generation tool, **`bison`**?

(Start variable is $\mathbf{E}$)

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{T} \mid \mathbf{T}$$
$$\mathbf{T} \rightarrow \mathbf{T} * \mathbf{F} \mid \mathbf{F}$$
$$\mathbf{F} \rightarrow \mathbf{num} \mid \mathbf{id}$$

Getting Started: Writing `.y` File

- **We can write `*.y` file to provide parser specification**
  - In the upper part, declare tokens and start symbol
  - In the lower part, define the production rules

`example.y` file

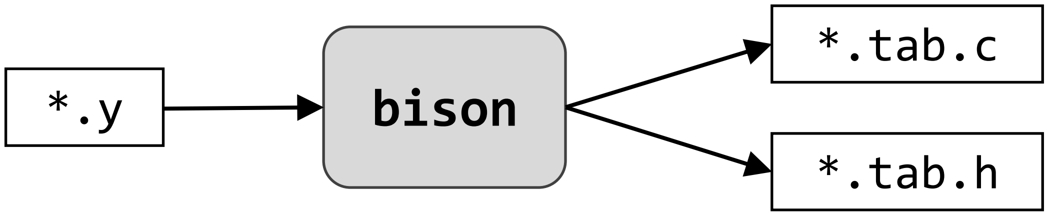
```
%union {
    int n;
    char *s;
}
%token <n> NUM
%token <s> ID
%token PLUS MULT
%start E

%%
E: E PLUS T
 | T ;
T: ... (omitted)
```

Declare that **`NUM`** token will be associated with an **`int`** type value

Getting Started: Running `Bison`

- **Now run `bison -d example.y` in the terminal**
  - It may seem awkward that **parser generation** takes place before **lexer generation**, but this is totally fine
- **It will generate `example.tab.c` and `example.tab.h`**
  - **`example.tab.c`** contains the generated parser code
  - **`example.tab.h`** contains the token declaration, and this will be used in the next lexer generation step

```
jschoi@cspro2:~/example$ bison -d example.y
```



Getting Started: Example RegEx

- **Let’s assume the following RegEx for each token**
  - Again, note that we used a similar RegEx in the lecture
  - How can we express this RegEx in lexer generation tool, **`flex`**?
  - Also, how can we tell the lexer to associate a concrete integer value with **NUM** token?

```
// Auxiliary RegEx
digit = [ '0'-'9' ]
letter = [ 'A'-'Z' ] | [ 'a'-'z' ]

// Actual tokens for our language
NUM = digit+
ID = (letter | '_') ( letter | digit | '_')*
PLUS = '+'
MULT = '*'
WHITESPACE = (' ' | '\t' | '\n')+
```

Getting Started: Writing `.l` File

- **We can write `*.l` file to provide lexer specification**
  - At the beginning of the `.l` file, include the header file previously generated by **`bison`**

`example.l` file

```
%{
#include "example.tab.h"
%}

%%

%%
```

Here, include previously generated header file

RegEx declaration will come here (next page)

RegEx definition for token will come here (next page)

Getting Started: Writing `.l` File

- **We can write `*.l` file to provide lexer specification**
  - In the upper part, declare auxiliary RegEx (like **`dig`**, **`let`**)
  - In the lower part, define the actual tokens in RegEx

`example.l` file

Here, write C code to specify which token must be returned

```
...
dig [0-9]
let [A-Za-z]

%%
"+"                           { return PLUS; }
"*"                           { return MULT; }
{dig}+                        { yylval.n = atoi(yytext);
                                return NUM; }
({let}|_)({let}|{dig}|_)*     { yylval.s = strdup(yytext);
                                return ID; }
[ \t\n]+                      { /* Skip (nothing to do) */ }
```

Getting Started: Writing `.l` File

- **Also, we can tell the lexer to associate certain value**
  - Here, **`yylval`** and **`yytext`** are pre-defined keywords (variables)
  - **`yytext`** is the string that has matched the left-side RegEx

`example.l` file

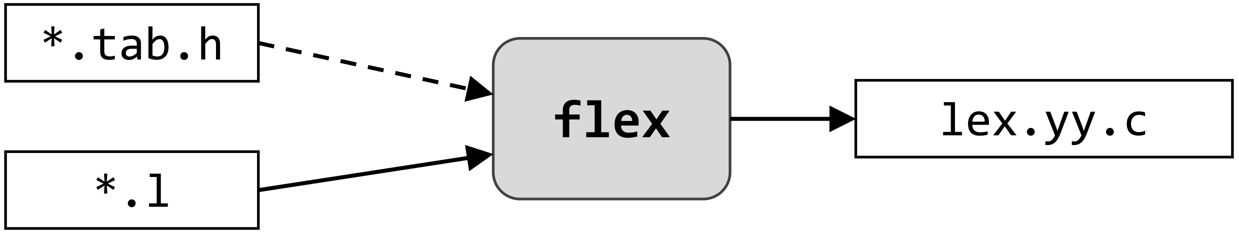
```
...
dig [0-9]
let [A-Za-z]

%%
"+"                       { return PLUS; }
"*"                       { return MULT; }
{dig}+                    { yylval.n = atoi(yytext);
                            return NUM; }
({let}|_)({let}|{dig}|_)* { yylval.s = strdup(yytext);
                            return ID; }
[ \t\n]+                  { /* Skip (nothing to do) */ }
```

Before returning **NUM** token, store the actual integer value

Getting Started: Running `Flex`

- Now run `flex example.l` in the terminal
- It will generate `lex.yy.c` file
  - The output file name is fixed as **`lex.yy.c`**
  - It contains the generated lexer code
- Note that `bison` must be executed before `flex`
  - **`*.tab.h`** file must be prepared before you run **`flex`**

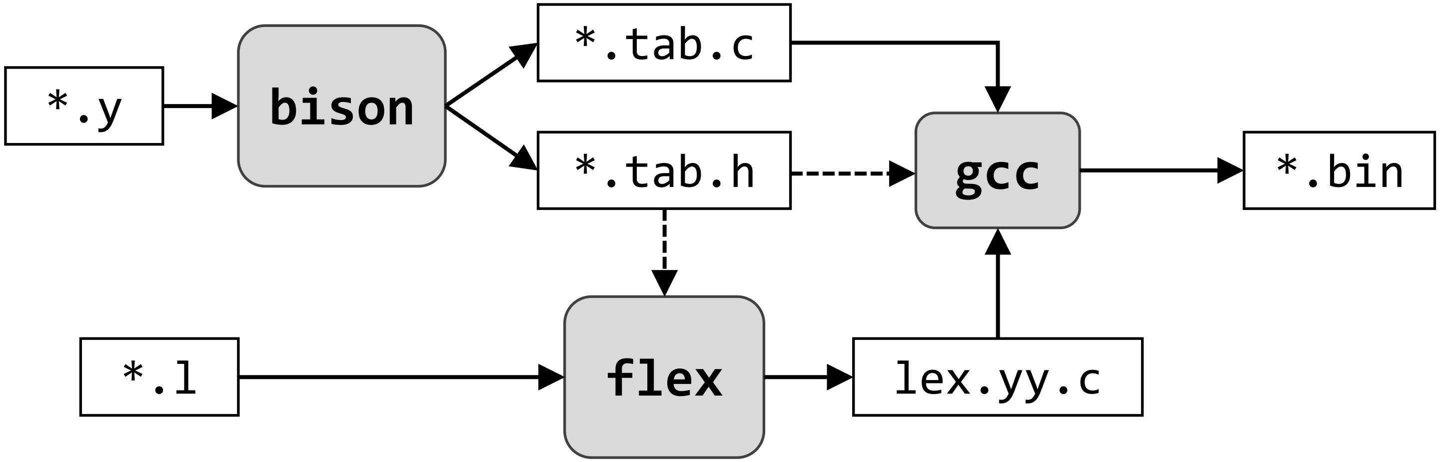
```
jschoi@cspro2:~$ flex example.l
```



Putting Things Together

- **Now, we can put all the things together and obtain a running implementation of front-end**
  - Take a look at the overall flow of input and output

```
jschoi@cspro2:~$ bison -d example.y
jschoi@cspro2:~$ flex example.l
jschoi@cspro2:~$ gcc -o main.bin example.tab.c lex.yy.c
```



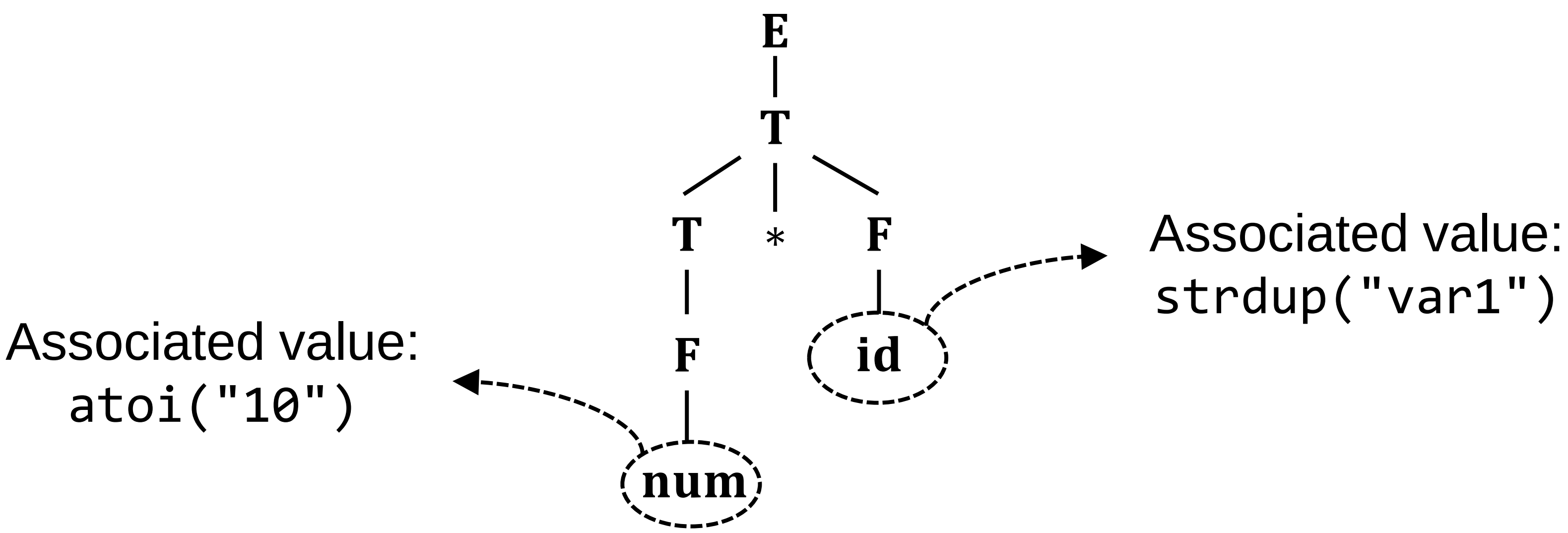
Now the front-end runs, but…

- **The compiled `main.bin` program will accept valid expressions and reject invalid ones**
  - The files are attached as **`Ex.tgz`**, so try it yourself
- **But it does not do anything interesting**
  - If we can obtain an AST from the input string, we may do more interesting things (such as computing the value of expression)

```
jschoi@cspro2:~$ cat tc-1
10 * var1
jschoi@cspro2:~$ ./main.bin tc-1
jschoi@cspro2:~$ cat tc-2
7 * + 3
jschoi@cspro2:~$ ./main.bin tc-2
error: syntax error
```

From Parse Tree to AST

- **Assume that the input string is `10 * var1`**
  - At token level, this will be represented as **num** ∗ **id**
- **Our current parser will generate the parse tree below**
  - Also, terminal nodes are associated with concrete values
  - Now, let’s fix the parser to generate **AST** from this parse tree

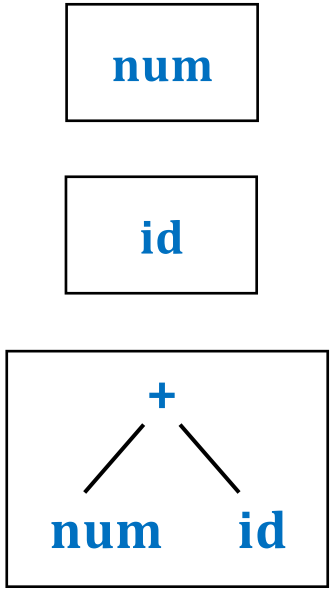


Representing AST in Code

- **First, let's define a C `struct` that represents an AST**
  - You will see similar **`struct`** definition in the skeleton code
  - Using this type, we can represent various ASTs

```c
// Represents the kind of expression.
typedef enum { NUM, ID, ADD, ... } AST_KIND;

struct AST {
  AST_KIND kind;
  int num;
  char *id;
  struct AST *left;
  struct AST *right;
};
```

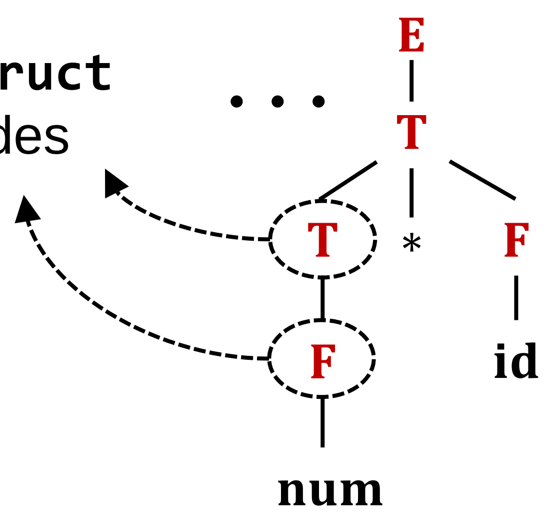
AST Examples



Adding Values to Non-terminals

- **As we did in `.l` file, we can fix `.y` file and specify the action to perform when each production rule is applied**
  - This action will compute a value and associate it with the parse tree's node for variable (non-terminal symbol)
  - On each node, we will add an **AST `struct`** that we defined
  - The **`struct`** added to **E** node will be the final AST that we want

Let's add AST **`struct`** on all these nodes



Action on CFG Rule: Example (1)

- Ex) Action for production rule **F → num**
  - **$$** means the value of node for left-side symbol, **F**
  - **$1** means the value of node for the 1st symbol on the right side (in this example, the 1st symbol is terminal **num**)

example.y file

```
%union {
    int n;
    char *s;
    struct AST *a;
}
...
%type <a> F

%%
...
F: NUM  { $$ = make_num_ast($1); }
 | ...
```

Declare that variable (non-terminal) **F** is associated with an **AST***

When this rule is applied, call **AST* make_num_ast(int n)**

Action on CFG Rule: Example (2)

- **Ex) Action for production rule T → F**
  - **$$** means the value of node for left-side symbol, **T**
  - **$1** means the value of node for the 1st symbol on the right side (this time, the 1st symbol is variable **F**)

example.y file

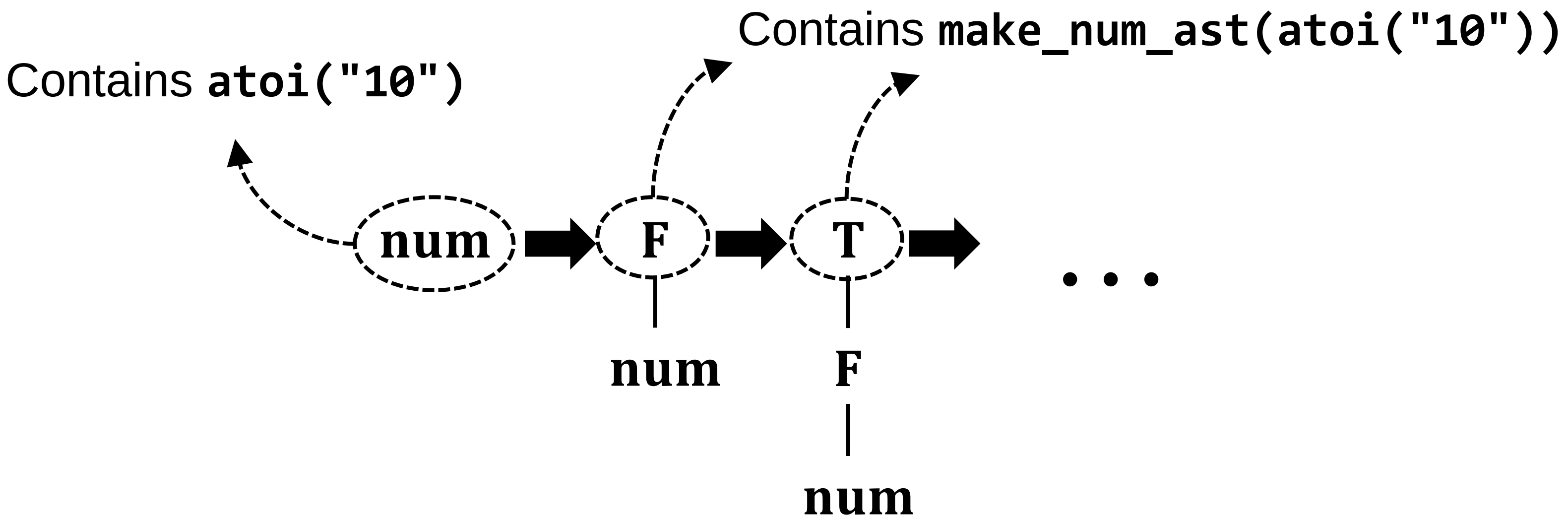
```
%union {
    int n;
    char *s;
    struct AST *a;
}
...
%type <a> T

%%
...
T: F    { $$ = $1; }
 | ...
```

T must be also associated with an **AST***

When this rule is applied, we can just copy the AST of **F** and add it to **T**

Example of AST Construction

- Now, we can imagine the overall process of parse tree and AST construction with input string `10 * var1`
- When $\mathbf{F} \rightarrow \mathbf{num}$ is applied (in other words, when **num** is reduced into **F**), `$$ = make_num_ast($1)` is performed
- Similarly, when $\mathbf{T} \rightarrow \mathbf{F}$ is applied, the AST stored in **F** will be copied and added to **T**



So far, we have covered the basics of `flex` and `bison`

Now let's move on to the actual skeleton code of our assignment

Our Source Language

- **You are going to write a front-end for simple numeric expression language**
  - You can optionally define variables in the first line
  - Your lexer/parser must be able to handle the examples below
  - Exact language specification will be given few pages later
- **Also, you will write code that computes the integer value denoted by the expression**
  - In other words, you will implement a simple ***calculator***

```
7 + 3 * 4
```

19

```
v_1 = 10;
-2 * v_1
```

-20

```
x = 8, y = 3;
(x - 2) / y
```

2

Skeleton Code

- **Copy `HW1.tgz` into CSPRO server and decompress it**
  - You **must connect to** `csproN.sogang.ac.kr` **(N = 2, 3, or 7)**
  - Don't decompress-and-copy; copy-and-decompress
- **`Makefile`: You can build the project by typing `make`**
- **`src/`: Source files you have to work with**
- **`testcase/`: Sample test case and their answers**
- **`check.py`: Script for self-grading with test cases**
- **`config`: Used by the grading script (you can ignore)**

```
jschoi@cspro2:~$ tar -xzf HW1.tgz
jschoi@cspro2:~$ ls HW1/
Makefile  check.py  config  src  testcase
```

Structure of `src` Directory

- `Makefile`: **The top-level `Makefile` redirects to this**
- `prog.l` **: Input file for `Flex` (*RegEx* + α)**
- `prog.y` **: Input file for `Bison` (CFG + α)**
- `ast.*` **and** `variable.*` **: C header and source files to help the implementation of front-end and calculator**
  - **`ast.*`** contains the code related to AST construction
  - **`variable.*`** contains the code related to variable initialization

```
jschoi@cspro2:~/HW1$ cd src/
jschoi@cspro2:~/HW1/src$ ls
Makefile  ast.c  ast.h  prog.l  prog.y  variable.c  variable.h
```

Source Language: Lexer Spec

- **Translate the following descriptions into regular expressions in `prog.l` file**
  - Most parts are already done for you
- **Your lexer must recognize each of these symbols as a token: "`+`" "`-`" "`*`" "`/`" "`=`" "`(`" "`)`" "`,`" "`;`"**
- **`ID` token can contain lower/uppercase alphabets, or digit characters (`0...9`), or underbar (_)**
  - But it cannot start with a digit character
- **`NUM` token can contain any digit characters (`0...9`)**
  - It does not include sign prefix
  - Therefore, string "**`-1523`**" must be recognized as two tokens

Source Language: Parser Spec

- **Context-free grammar for our source language**
  - $\mathbf{Prog \rightarrow\ E \mid Vars\ ;\ E}$
  - $\mathbf{Vars \rightarrow id\ = num \mid id\ = num,\ Vars}$
  - $\mathbf{E \rightarrow E + T \mid E - T \mid T}$
  - $\mathbf{T \rightarrow T * F \mid T \,/\, F \mid F}$
  - $\mathbf{F \rightarrow num \mid id \mid (\ E\ ) \mid\ - F}$
- **Specify this CFG in `prog.y`**
  - Some rules are already implemented for you
- **Ask me if this CFG seems to have a problem or mistake**

Your Mission

- **Complete `prog.l` and `prog.y` according to the spec**
  - **But do not change** `main()` and `yyerror()` code in `prog.y`
- **You also have to implement some C functions in `ast.c` (functions marked with "`TODO`" comment)**
- **Do not touch any other files**
  - Such as `Makefile`, `ast.h`, and `variable.*` files
- **(Tip) Before you start, take enough time to read and understand the skeleton code**
  - Once you understand the code, this assignment is easy

Self-Grading

- **In `testcase/` directory, `tc-N` (test input) and `ans-N` (expected output of `tc-N`) files are provided**
- **You can run a test case with a command like this:**

  `./main.bin testcase/tc-N`

- **You can also use `check.py` to run all the test cases**
  - **Meaning of each character:** `'O'`: Correct, `'X'`: Incorrect, `'T'`: Timeout, `'E'`: Runtime error, `'C'`: Compile error

```
jschoi@cspro2:~/HW1$ ./main.bin testcase/tc-1
19
jschoi@cspro2:~/HW1$ ./main.bin testcase/tc-2
0
jschoi@cspro2:~/HW1$ ./check.py
[*] Result : OXXX
```

Test Cases for Real Grading

- **During the real grading, I will use additional test cases**
  - So you are strongly encouraged to run your own test cases
- **Assumptions for test cases**
  - All the variables that appear in the numeric expression are properly initialized (no uninitialized variable)
  - Each variable is initialized only once (no duplicate initialization)
  - You don't have to consider large integer values for `NUM` tokens; test cases will only contain integers between 0 to 65535
  - Invalid inputs (inputs rejected by front-end) will not be used
    - You don't have to worry about reporting lexical/syntax errors

Submission Guideline

- **You should submit the following three files**
  - `prog.l`
  - `prog.y`
  - `ast.c`
- **No report is required for this assignment**
- **Submission format**
  - Upload these files directly to *Cyber Campus* (**do not zip them**)
  - **Do not change the file name** (e.g., adding any prefix or suffix)
  - If your submission format is wrong, you will get **-20% penalty**