Programming Assignment #1. Flex & Bison Exercise

Prof. Jaeseung Choi

Dept. of Computer Science and Engineering

Sogang University

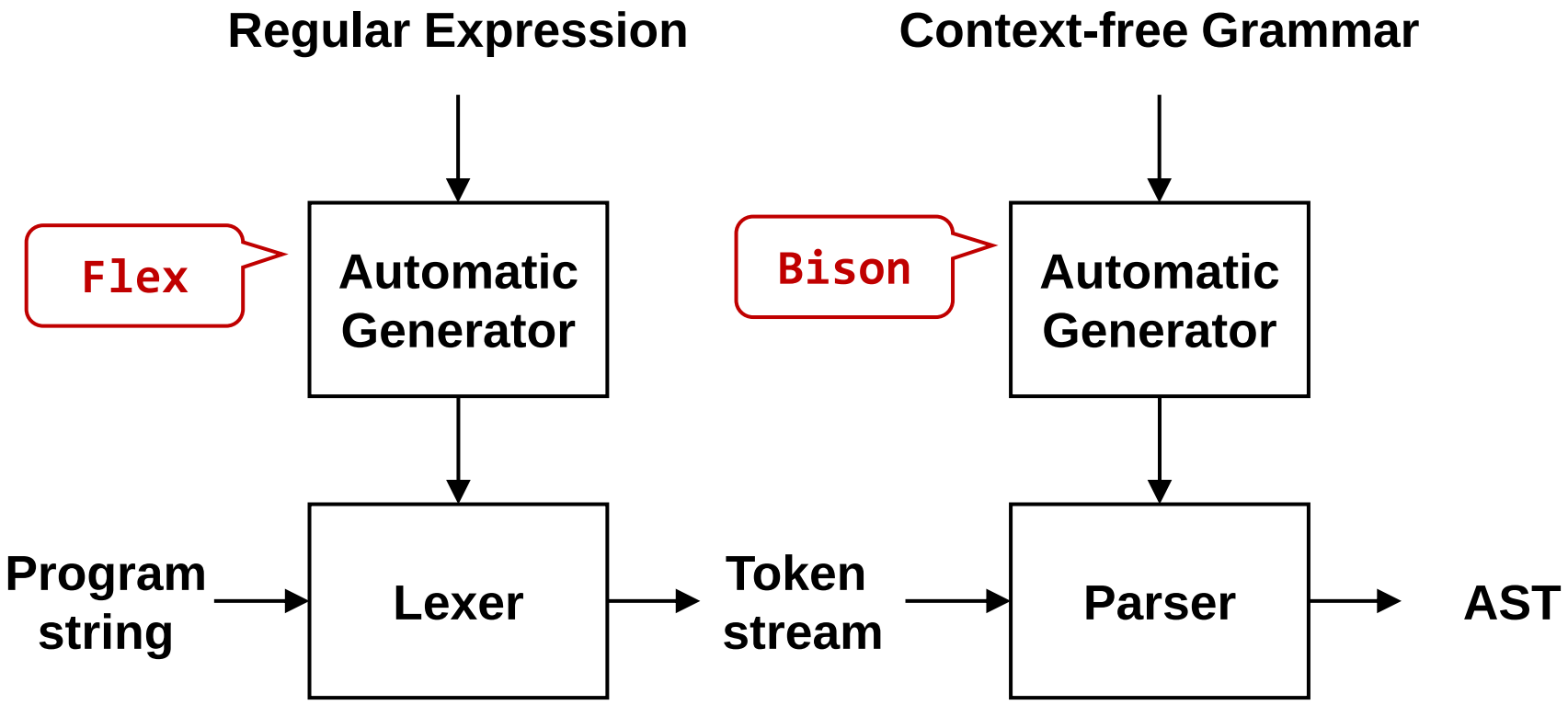
General Information

- **Check "HW #1" in *Assignment* tab of *Cyber Campus***
  - Skeleton code (`HW1.tgz`) is attached in the post
  - Deadline: **10/16** Mon. 23:59
  - Submission will be accepted in that post, too
  - Late submission deadline: **10/18** Wed. 23:59 **(-20% penalty)**
  - Delay penalty is applied uniformly **(not problem by problem)**
- **Please read the instructions in this slide carefully**
  - This slide is 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

Remind: Cheating Policy

- **Cheating (code copy) is strictly forbidden in this course**
  - Read the orientation slide once more
- **Don’t ask for solutions in the online community**
  - TA will regularly monitor the communities
- **Sharing your code with others is as bad as copying**
  - Your cooperation is needed to manage this course successfully

Automatic Front-End Generation



Flex/Bison vs. Lex/Yacc?

- **Flex (Fast Lex) is rewritten version of Lex**
  - Also, the license is more permissive
- **Similarly, Bison is rewritten version of Yacc**
- **We will use Flex and Bison for this assignment**
- **In most Linux systems, Lex/Yacc are automatically redirected to Flex/Bison**
  - Still, many people just call them **Lex** and **Yacc**

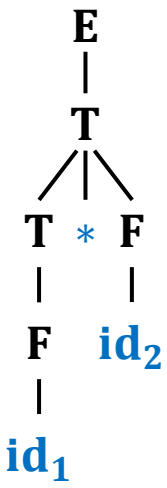
`Bison`: LALR Parsing

- **`Bison` generates an LALR parser for the provided CFG**
- **Recall that LALR is one kind of *bottom-up* parsing**
- **Although we have not learned the details of bottom-up (and LALR) parsing, we can still use `Bison`**
  - All you have to know is how to deal with CFG
  - But after finishing the ***Syntax Analysis*** chapter, you will better understand what is internally going on

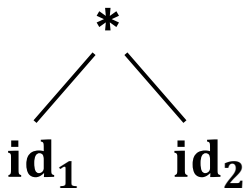
Goal of This Assignment

- **Of course, the first goal is learning how to use `Flex` and `Bison` to generate front-end code**
- **Another important topic to cover is constructing AST**
  - In ***Syntax Analysis*** chapter, we focused on **parse tree**
  - However, **AST** is more concise and desirable form
  - Our `Bison`-generated parser will construct AST *on the fly*

Parse tree of $\mathbf{id_1 * id_2}$

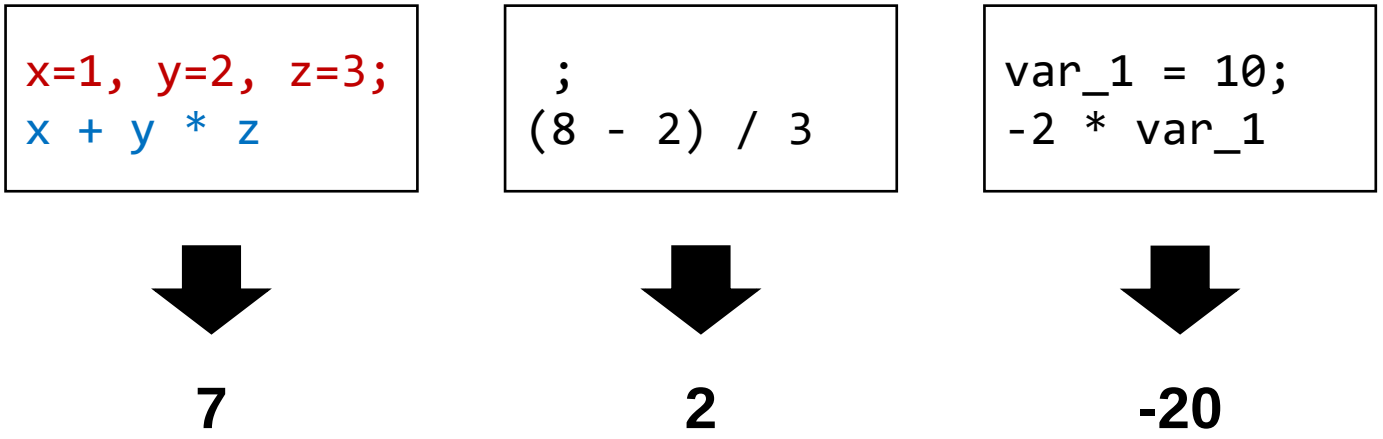


Abstract syntax tree (AST) form



Our Source Language

- **We are going to write a front-end for simple numeric expression language**
  - Initialize variables with values (optional)
  - Initialization is followed by a single numeric expression
- **Our lexer/parser must be able to handle programs below**
  - Also, we will compute the value denoted by the expression



Skeleton Code

- **Copy HW1.tgz into CSPRO server and decompress it**
  - Don’t decompress-and-copy; copy-and-decompress
  - This course will use cspro**5**.sogang.ac.kr (**don’t miss the "5"**)
- **`src/`: Source files you have to work with**
- **`Makefile`: Type `make` to build the whole project**
  - Internally redirects to **`src/Makefile`**
- **`testcase/`: Sample test cases and their answers**
- **`check.py`: Script for self-grading with test cases**
- **`config`: Used by the grading script (you don’t have to care)**

```
jason@ubuntu:~$ tar -xzf HW1.tgz
jason@ubuntu:~$ ls HW1
check.py  config  Makefile  src  testcase
```

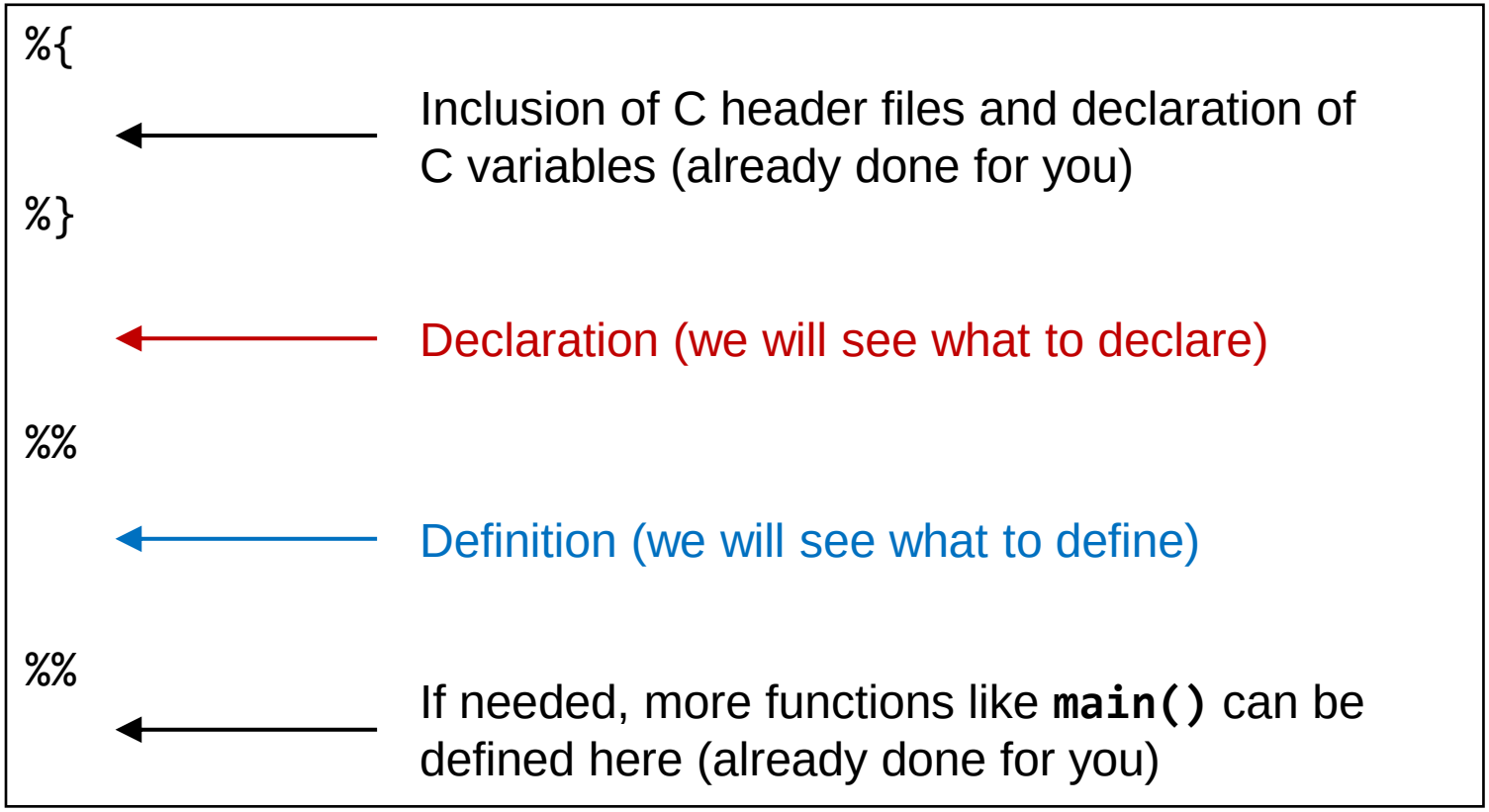
`Src` Directory Structure

- **First, you must understand the provided code**
  - Once you know what is going on, this HW is not much hard
  - You only have to write or fix about 50+ lines of code
- **`prog.l` : Input file for `Lex/Flex` (*RegEx* + α)**
- **`prog.y` : Input file for `Yacc/Bison` (CFG + α)**
- **`ast.*` and `varlist.*` : C header and source files to help the implementation of our front-end**
- **`Makefile`: The top-level `Makefile` redirects to this one**

```
jason@ubuntu:~/CSE4120-Lab/HW1$ cd src
jason@ubuntu:~/CSE4120-Lab/HW1/src$ ls
ast.c  ast.h  Makefile  prog.l  prog.y  varlist.c  varlist.h
```

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

- **Both `Lex` file (`.l`) and `Yacc` file (`.y`) have the following structure**



Getting Started: My First `.y` File

- **We will start with a simple CFG (not the complete one)**
  - Caution: the code below is **NOT** the skeleton code
- **First, declare tokens and start symbol in `prog.y`**
- **Then, define the production rules of CFG in `prog.y`**

Simplified CFG

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow id \mid num$$

```
%token NUM ID PLUS MULT
%start Exp

%%

Exp:
 Term
 | Exp PLUS Term
;
...
```

Getting Started: Run `Bison`

- **Now run "`bison -d prog.y`" on the completed `Yacc` file**
  - It may seem awkward to generate a parser even before a lexer is prepared, but this is totally fine
- **It will generate `prog.tab.c` and `prog.tab.h`**
  - **`prog.tab.c`** is the generated parser code
  - **`prog.tab.h`** will be included by **`prog.l`** to use token declaration

The first part of **`prog.l`** file (continued in the next page)

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

...
```

Getting Started: My First `.l` File

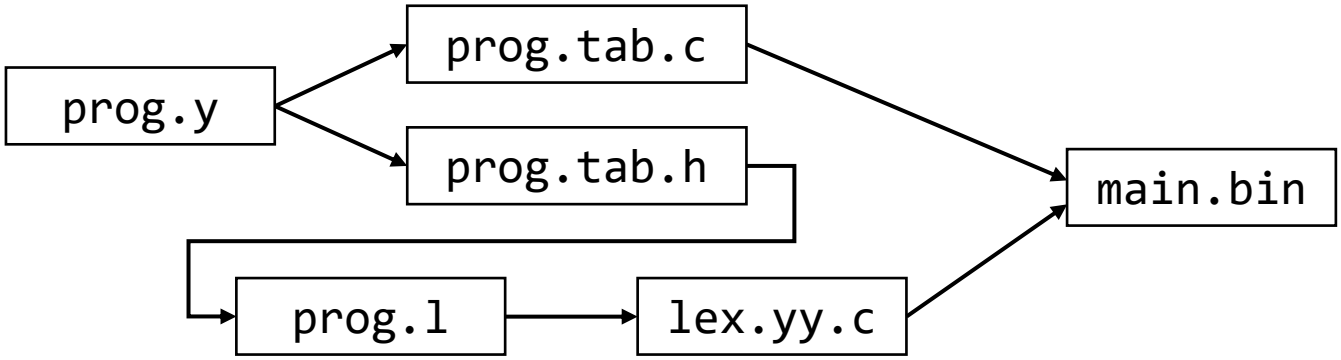
- **Now, define the *RegEx* for each token in `prog.l`**
  - Notation is slightly different from the ***Lexical Analysis*** Chapter
  - We specify in C which token to return for each *RegEx* pattern
    - If the code snippet doesn't return anything, token is skipped
  - We can declare some patterns (**`dig`**, **`let`**) to remove duplication

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

%%

"+"                             { return PLUS; }
"*"                             { return MULT; }
{dig}+                          { return NUM; }
({let}|_)({let}|{dig}|_)*       { return ID; }
[ \t\n]+                        { /* Skip (do nothing) */ }
```

Putting Things Together

- **Now run "`flex prog.l`" on the completed `Lex` file**
  - It will generate the lexer code in **`lex.yy.c`** file
- **Finally, compile the generated code with `gcc`**
  - A simple **`Makefile`** will look like this

```
../main.bin: prog.y prog.l
        bison -d prog.y
        flex prog.l
        gcc -o ../main.bin prog.tab.c lex.yy.c
```



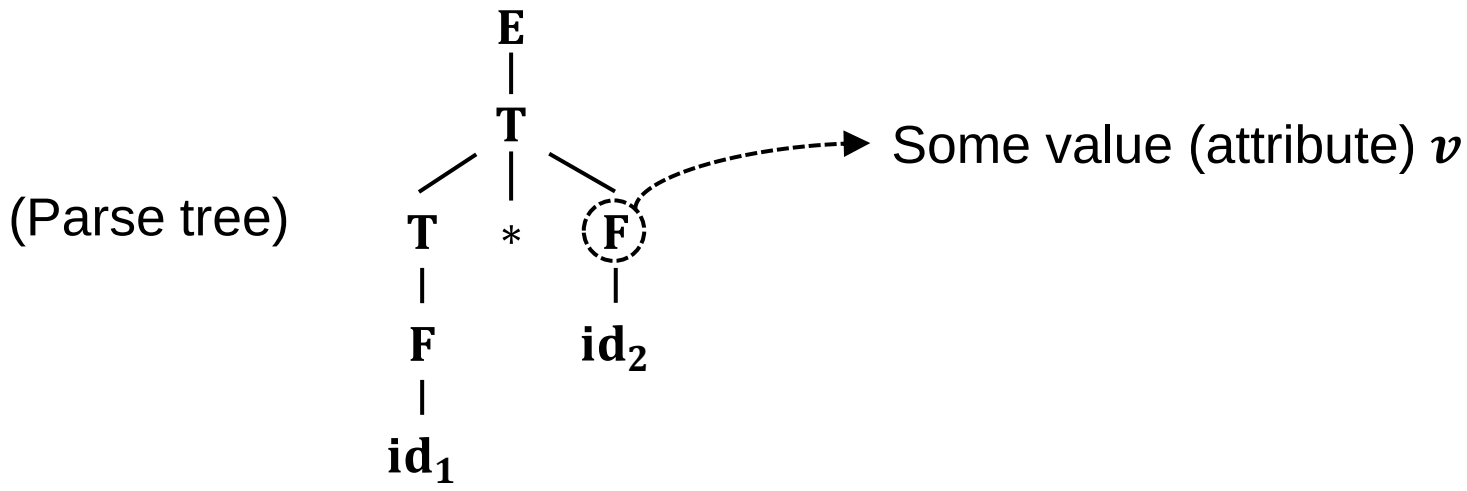
Now the front-end runs, but…

- The compiled `main.bin` will accept valid numeric expression and reject invalid ones
- But it does not do anything interesting
- Let's fix our front-end to construct AST and evaluate the value of numeric expression

```
$ cat tc-1
7 + 3 * 4
$ ./main.bin tc-1
$ cat tc-2
7 + 3 * + 4
$ ./main.bin tc-2
error: syntax error
```

Adding Values and Actions

- **The key idea is to associate some value (a.k.a. attribute) with each node of parse tree**
- **We also define appropriate actions for each CFG rule**
  - Each action is executed when the rule is applied
- **These actions compute value (attribute) for each node**



Values for Terminal Symbol (Token)

- **We must store the value of a token to `yylval`**
  - `NUM` token must be associated with an integer
  - `ID` token must be associated with a string (pointer)
- **`yytext` is the string that just has been matched by lexer**

prog.y

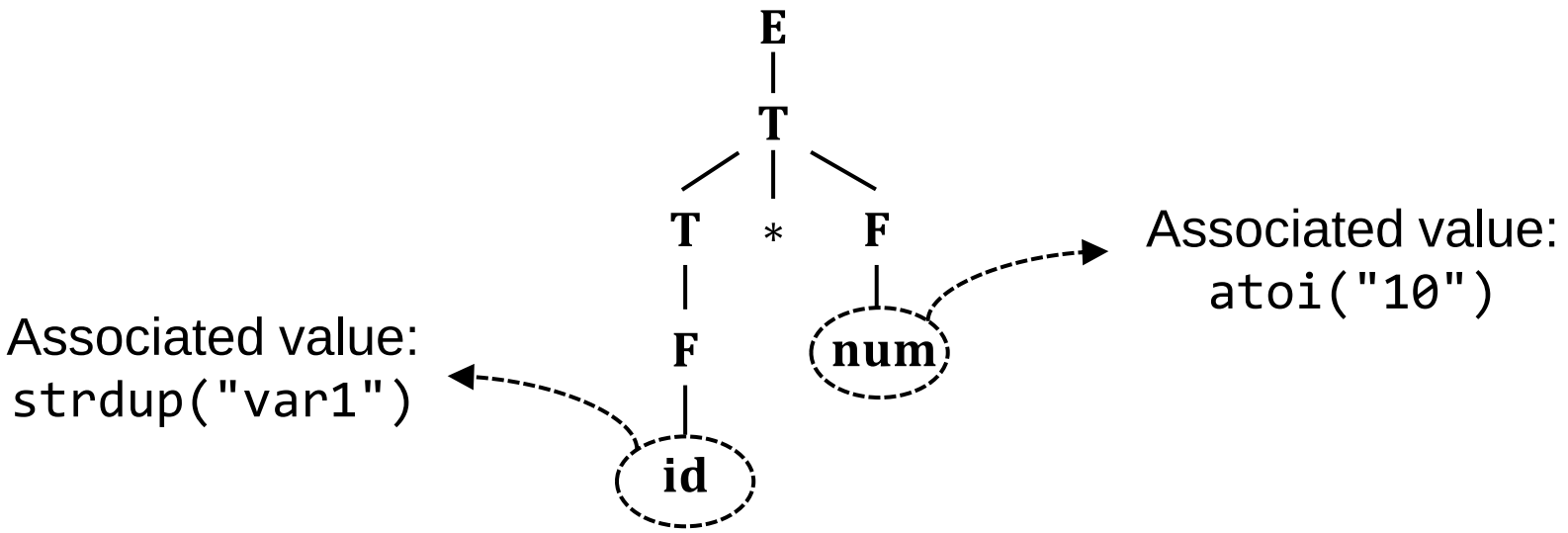
```
%union {
    int n;
    char *s;
}
%token <n> NUM
%token <s> ID
%token PLUS MULT
...
%%
...
```

prog.l

```
"+"     { return PLUS; }
"*"     { return MULT; }
{dig}+ {
    yylval.n = atoi(yytext);
    return NUM;
}
({let}|_)({let}|{dig}|_)*   {
    yylval.s = strdup(yytext);
    return ID;
}
```

Terminal's Value: Example

- **Assume that the input string is `var1 * 10`**
  - At token level, represented as **id * num**
  - Value of **`yytext`** is `"var1"` for **id** token, `"10"` for **num** token
- **The values of terminal symbols are illustrated below**
- **Now, how to compute values for non-terminals (E, T, F)?**



Values for Non-terminal Symbol

- **Example: Actions for the rules $F \rightarrow num$ and $F \rightarrow id$**
  - ***$$*** is the value of ***LHS*** symbol, ***$n*** is the value of ***RHS*** symbols

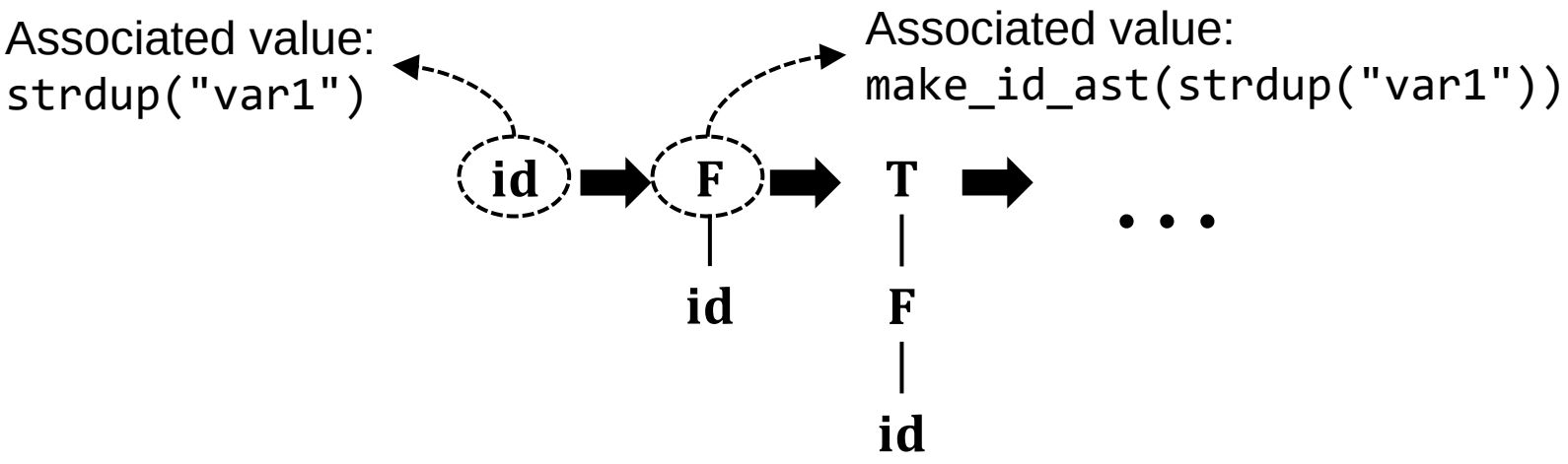
```
%union {
    int n;
    char *s;
    struct AST *a;
}
...
%type <a> Fact

%%
...
Fact:
 NUM     { $$ = make_num_ast($1); }
 | ID    { $$ = make_id_ast($1); }
;
```

Declare that non-terminal symbol **Fact** is associated with a **AST***

AST* make_num_ast(int n) and **AST* make_id_ast(char *s)** are defined in **ast.h/ast.c** files

Non-terminal's Value: Example

- **Again, assume input string `var1 * 10`**
- **Recall the bottom-up parsing process for this string**
- **In the first step that applies rule $F \rightarrow id$ reversely, its action `$$ = make_id_ast($1)` is performed**
  - **`$1`** is the value associated with the first symbol in RHS (**id**)
  - Recall that we have previously set it with **`strdup("var1")`**



Now basic explanation is over

You must read the skeleton code and figure out what is going on (it will take some time)

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 these symbols as distinct tokens: "`+`", "`-`", "`*`", "`/`", "`=`", "`(`", "`)`", "`,`", "`;`"**
- **Identifier token can start with any alphabet or _, and trailing characters can be alphabet, digit, or _**
- **Number token can be any decimal integer**
  - 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**
  - $\mathbf{Program} \rightarrow \mathbf{;}\ \mathbf{Exp} \mid \mathbf{Init}\ \mathbf{;}\ \mathbf{E}$
  - $\mathbf{Init} \rightarrow \mathbf{id}\ = \mathbf{num} \mid \mathbf{id}\ = \mathbf{num},\ \mathbf{Init}$
  - $\mathbf{E} \rightarrow \mathbf{E} + \mathbf{T} \mid \mathbf{E} - \mathbf{T} \mid \mathbf{T}$
  - $\mathbf{T} \rightarrow \mathbf{T} * \mathbf{F} \mid \mathbf{T} / \mathbf{F} \mid \mathbf{F}$
  - $\mathbf{F} \rightarrow \mathbf{num} \mid \mathbf{id} \mid (\ \mathbf{E}\ ) \mid -\mathbf{F}$
- **Specify this CFG in `prog.y`**
  - Some rules are already implemented for you
- **Symbol names do not have to exactly match with this**
  - The order of rules does not matter, too
- **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` and `varlist.c` (functions marked with `"TODO"` comment)**
- **You will be asked to submit these four files**
- **Do not touch any other files**
  - Such as **`Makefile`** or header files (**`ast.h`** and **`varlist.h`**)

Self-Grading

- **Run `check.py` script to run your code with test inputs in `testcase/` directory**
  - Symbols in the result have the following meanings
    - **`'O'`**: Correct, **`'X'`**: Incorrect
    - **`'T'`**: Timeout, **`'E'`**: Runtime error, **`'C'`**: Compile error
  - In **`testcase/`** directory, **`tc-N`** (test input) and **`ans-N`** (expected output of **`tc-N`**) files are provided

```
jason@ubuntu:~/HW1$ ls
check.py  config  main.bin  Makefile  src  testcase
jason@ubuntu:~/HW1$ ./check.py
[*] Result: XXXX
```

Test Cases for Real Grading

- **During the real grading, I will use additional test cases**
- **So you are 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)
  - I will not use invalid inputs as test cases
    - You don’t have to worry about reporting lexical/syntax errors

Submission Guideline

- **You should submit the following four files**
  - `prog.l`
  - `prog.y`
  - `ast.c`
  - `varlist.c`
- **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**