Compiler Construction

Lab 2

Sept 23, FSU/CS

Your language is defined as before

$$
\begin{aligned}
\mathit{program} &\rightarrow (\texttt{define-fun}\ (\mathit{fun}\ (\mathit{type}\ \mathit{var})^*)\ \mathit{type}\ \mathit{expr})\ \mathit{program} \mid (\texttt{eval}\ \mathit{expr}) \\
\mathit{type} &\rightarrow \texttt{int} \mid \texttt{bool} \\
\mathit{expr} &\rightarrow \mathit{term} \mid \mathit{fla} \\
\mathit{term} &\rightarrow \mathit{const} \mid \mathit{var} \mid (\texttt{get-int}) \mid (\texttt{+}\ \mathit{term}\ \mathit{term}^+) \mid (\texttt{*}\ \mathit{term}\ \mathit{term}^+) \mid (\texttt{-}\ \mathit{term}\ \mathit{term}) \mid \\
&\quad (\texttt{div}\ \mathit{term}\ \mathit{term}) \mid (\texttt{mod}\ \mathit{term}\ \mathit{term}) \mid \\
&\quad (\texttt{if}\ \mathit{fla}\ \mathit{term}\ \mathit{term}) \mid (\mathit{fun}\ \mathit{expr}^*) \mid (\texttt{let}\ (\mathit{var}\ \mathit{expr})\ \mathit{term}) \\
\mathit{fla} &\rightarrow \texttt{true} \mid \texttt{false} \mid \mathit{var} \mid (\texttt{get-bool}) \mid \\
&\quad (\texttt{=}\ \mathit{term}\ \mathit{term}) \mid (\texttt{<}\ \mathit{term}\ \mathit{term}) \mid (\texttt{<=}\ \mathit{term}\ \mathit{term}) \mid (\texttt{>}\ \mathit{term}\ \mathit{term}) \mid (\texttt{>=}\ \mathit{term}\ \mathit{term}) \mid \\
&\quad (\texttt{not}\ \mathit{fla}) \mid (\texttt{and}\ \mathit{fla}\ \mathit{fla}^+) \mid (\texttt{or}\ \mathit{fla}\ \mathit{fla}^+) \mid (\texttt{if}\ \mathit{fla}\ \mathit{fla}\ \mathit{fla}) \mid \\
&\quad (\mathit{fun}\ \mathit{expr}^*) \mid (\texttt{let}\ (\mathit{var}\ \mathit{expr})\ \mathit{fla})
\end{aligned}
$$

Your task

Resolve all conflicts for *var* and *fun*:

- Redesign your productions (e.g., by moving *term* and *fla* to be *expr* and unifying their productions)
- Your productions will allow syntactic constructs that are not well-formed (e.g., `(< (< a b) z)`), but you will refine them by the semantic/type checks (see next page)

Implement a symbol table to keep track of:

- Defined functions, their arities, types of arguments, and return types
- Declared input variables and local variables used in *let* (derived types)

Implement an Abstract Syntax Tree (AST) generator

- AST should have the capabilities of pretty-printing to a graphviz file (https://graphviz.org/) which is further compilable to a PDF file
- If there are lexical/syntax/semantic errors

In the case of semantic errors,

- the compiler should skip the AST generation
- the compiler should print an error message with some description
- If there are two or more errors, it is OK to print description of only one of them

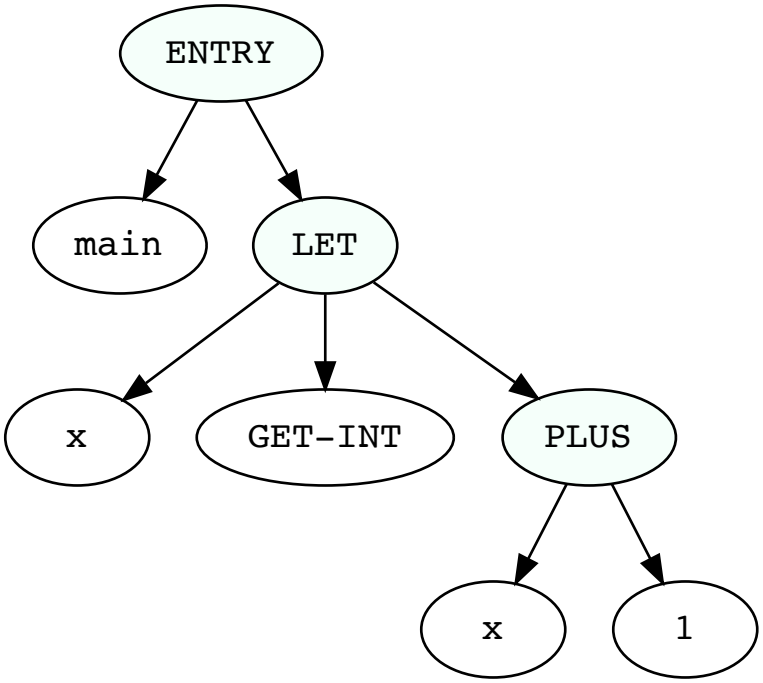
Your task (cont)

Implement the following semantic/type-checking passes using AST and symbol table:

- Well-formedness of all syntactic constructs w.r.t. original grammar rules (i.e., all arguments of an operator/function have the expected types)
- Variables can be used only if they are previously declared in the `define-fun` of the closest function or the `let`-binder
- Each variable name can be declared exactly once per each `define-fun` and/or nested `let`
- But variables with the same names can be used in two or more `let`-s if their parse trees don't overlap
- Functions can be called only if they are previously defined (except `get-int` and `get-bool`)
- Each function name can be used in exactly one `define-fun` and cannot be a variable name
- The number and types of arguments of each function call should match the number and types of arguments in the `define-fun` of the corresponding function
- The return type of each function should be consistent with the type of each its call
- The type of each `let` variable (i.e., `int` or `bool`) should be uniquely identifiable by the corresponding *expr*

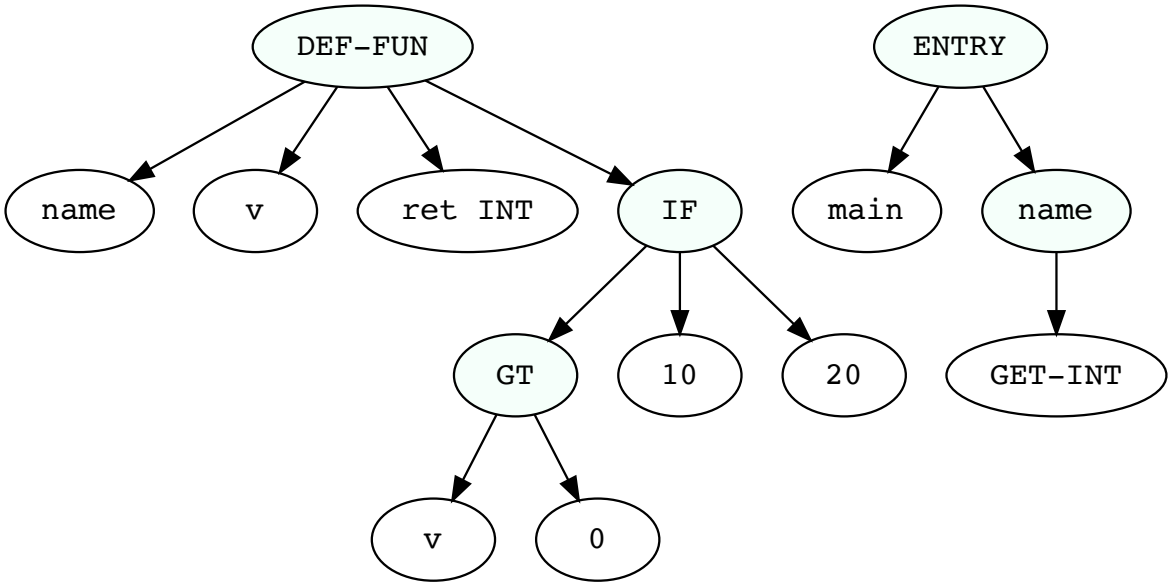
Example of correct program and AST

```
(eval (let (x (get-int)) (+ x 1)))
```



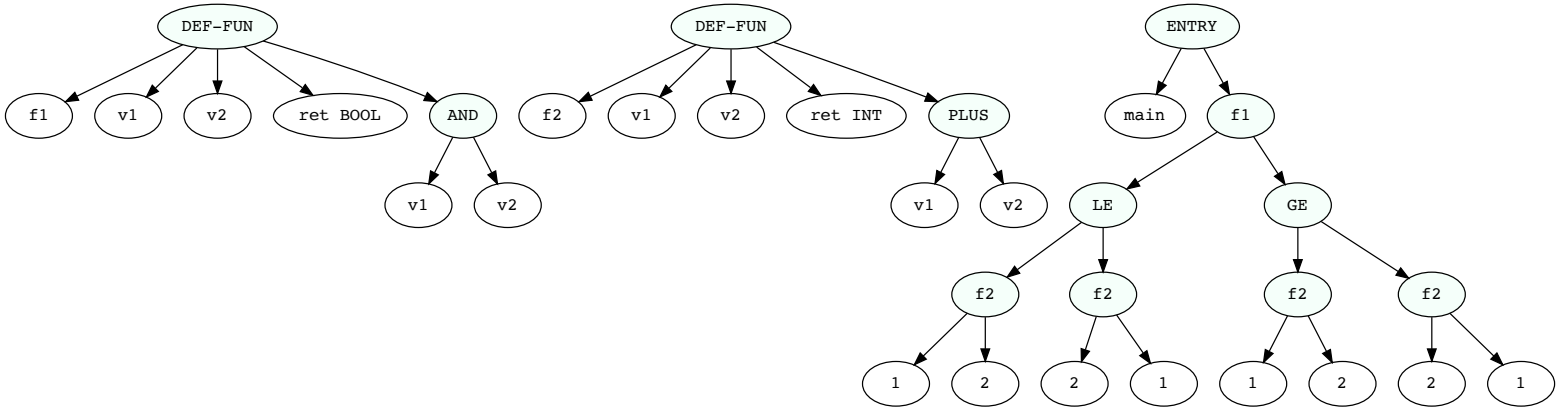
Example of correct program and AST

```
(define-fun (name (int v)) int (if (> v 0) 10 20))
(eval (name (get-int)))
```



Example of correct program and AST

```
(define-fun (f1 (bool v1) (bool v2)) bool (and v1 v2))
(define-fun (f2 (int v1) (int v2)) int (+ v1 v2))
(eval (f1 (<= (f2 1 2) (f2 2 1)) (>= (f2 1 2) (f2 2 1))))
```



Examples of incorrect programs

- `(eval (let (v true) (div v 2)))`
  Output: Argument #1 of `div` does not type check with type of `v`

- `(eval (notdefinedfunction 1 2 3))`
  Output: Function `notdefinedfunction` is not defined

- `(define-fun (foo (bool v)) int (if v 10 20))`
  `(eval (foo 1 2))`
  Output: Wrong number of arguments of function `foo`

- `(define-fun (f (int v)) int (let (v (+ 1 2)) (* v 2)))`
  `(eval (f 1))`
  Output: Variable `v` is declared twice

- `(define-fun (bfun (bool v)) int (if v 1 0))`
  `(eval (bfun 1))`
  Output: Argument #1 of `if` does not type check with type of `v`

- `(define-fun (fun1 (bool v2) (bool v3)) bool (or v2 v3))`
  `(define-fun (fun2 (bool v1)) bool (or v2 v3))`
  `(eval (fun2 true))`
  Output: Variable `v2` is not declared

Important

- The yacc file should only have the AST generation
- The semantic/type-checking passes should be implemented using the visitor pattern over the generated AST
- The repository should have Makefile
- The name of your binary should be `comp`
- Your code should be committed to your GitHub repository
- Test cases should be committed (both correct and incorrect programs)