School of Mathematical and Computational Sciences

Indian Association for the Cultivation of Science

Compiler Construction: COM 5202

Tutorial IX (26 March, 2025)

M. Sc Semester IV: 2024-2025 *Instructor:* Goutam Biswas

Exercise 1. *Consider the following grammar G of simple type declaration.*

$$\begin{aligned} D &\rightarrow T\ L \\ T &\rightarrow \texttt{int} \\ T &\rightarrow \texttt{float} \\ L &\rightarrow L\ ,\ id \\ L &\rightarrow id \end{aligned}$$

(a) *Transform it an equivalent $LL(1)$ grammar G_1 by removing left-recursion.*

(b) *The non-terminal T has a synthesized attribute $T.t$ to remember the type. Non-terminal L and L' (created after the removal of left-recursion) each has an inherited attribute, $L.i$ and $L'.i$ to propagate the type information. Give an SDD corresponding to the rules of G_1 so that type information can be updated in the symbol table.*
Assume that the scanner is posting the identifier(ID) in the symbol table and returns the pointer to the entry as the attribute of ID.

(c) *How do you pass the synthesized and inherited attributes in a recursive descent $LL(1)$ parser?*

Exercise 2. *The following grammar generates radix-3 positional numeral ($\{0, 1, .2\}$, positiion weights are power of 3) with a radix-3 point.*

$$\begin{aligned} S &\rightarrow L \cdot L \\ S &\rightarrow L \\ L &\rightarrow L\ B \\ L &\rightarrow B \\ B &\rightarrow 2 \\ B &\rightarrow 1 \\ B &\rightarrow 0 \end{aligned}$$

Design an SDD to compute $S{\cdot}val$, the decimal value of the input string. For example, the translation of `102.201` *should be the decimal number $1 \times 3^2 + 0 \times 3^1 + 2 \times 3^0 + \frac{2}{3} + \frac{0}{3^2} + \frac{1}{3^3} = 11\frac{7}{9} \approx 11.78$ in decimal.*

Exercise 3. **[10]**

In this assignment you will use a simple symbol table and do some basic context-sensitive analysis during the parsing phase. The recursive-descent parser of **Tutorial VII, Ex. 3** *(26 February, 2025) will be augmented with necessary C code.*

The suggested symbol table is a linked list (unordered) of the following type of structures. You may change the structure if you wish.

```
typedef struct sym {
        char *varP;
        struct sym *next;
        char type; // 'i': int, 'r': double, 'n': not assigned
        char init; // 't': initialized, 'f': not initialized
        union {
            int integer;
            double real;
        } val ;
} *symP;
```

Following is an explanation of different fields.

- `varP` - *pointer to the identifier name, the content of* `yylval.string` *when the token is* `ID`.
- `next` - *pointer to the next entry of the symbol table record.*
- `type` - *single character, the type of the variable. '*`i`*' for integer, '*`f`*' for floating-point number, and '*`n`*' - not known.*
- `init` - *flag whether the variable is initialized or not. '*`t`*' for initialized and '*`f`*' for uninitialized.*
- `val` - *data, union of* `int` *and* `double`.

Following are the actions to be taken:

(a) *The symbol table is a global data structure.*

(b) *Modify the scanner* `lex.l` *so that it will make an entry of an identifier in the symbol table if it is not already there.*
It will search for the identifier in the existing table. If it is not present, it creates a node of type `struct sym`*, inserts it in the existing table and initialises the following fields:*
`varP`*: with identifier name.*
`next`*: pointer to the existing table.*
`type`*: unknown.*
`init`*: not initialised.*

Finally the the scanner will return the token ID along with the pointer to the corresponding node in the symbol table.

(c) *The parser is augmented with the following actions.*

- *The type of a variable is updated in the symbol table when the type information is available.*
- *Decleration of the same variable twice is an error:*
  `zah a0, b0, c0`
  `flt b0`
- *A variable is initialized by two rules.*
  RC: `input fact` *and*
  AC: `fact = 2*n`

In both the cases appropriate flag of the symbol table entry for `fact` *is to be updated.*
It is an **error** *if the variable is not declared.*

- *An uninitialized variable can be detected when it is used in an expression F:* `ID`.
  It is an error in our case if a variable is used in an expression without initialization.
- *If it is an error, report it, but* **fix the error** *and continue parsing.*

Use the `Makefile` *of* **Tutorial 7, Ex. 3**.
Finally prepare a `.tar` *file using the following command.*
`$ tar cvf <roll no>.9.tar y.tab.h lex.l recursiveDescent.c Makefile`
Send the `.tar` *file to* `goutamamartya@gmail.com`.

Input

```
// Syntactically correct program
{
     zah  a0, b0, c0
 :
        input a0
        b0 = 5*a0
        c0 = b0 + b0/a0 - 10
        print c0
}
```

Output

```
$ ./a.out < d1
Accept
```

Input

```
// Same variable declared twice
{
     zah  a0, b0, c0, b0
 :
        input a0
        b0 = 5*a0
        c0 = b0 + b0/a0 - 10
        print c0
}
```

Output

```
$ ./a.out < d2
Redeclaration: b0: (3)
Reject
```

Input

```
// Var not declared and more
{
     zah  b0, c0
 :
        input a0
        b0 = 5*a0
        c0 = b0  b0/a0 - 10
        print c0
}
```

Output

```
$ ./a.out < d3
Variable a0 not decl.: (5)
'=' missing: (7)
```

Input

```
// Variable not initialized
{
     zah  a0, b0, c0
 :
        // input a0
        b0 = 5*a0
        c0 = b0 + b0/a0 - 10
        print c0
}
```

Output

```
$ ./a.out < d4
Variable a0 not init.: (6)
Reject
```