BITS Pilani, Hyderabad Campus
Department of Computer Science and Information Systems
Second Semester, 2024-25
CS F363 Compiler Construction
Lab-9: Abstract Syntax Tree generation

1 Objectives

The objectives of this lab sheet are given below.

- Gain a clear understanding of the purpose and structure of Abstract Syntax Trees (ASTs) as a crucial intermediate representation in the compilation process.
- Develop practical skills in using Bison and Flex to implement the construction of ASTs for various C programming language constructs, including expressions, assignments, conditionals, loops, and declarations.
- Learn to visually represent ASTs using tree diagrams and interpret the semantic meaning of source code by analyzing the structure of its corresponding AST.
- Extend the capabilities of a basic parser to handle more complex language features, thereby expanding understanding of compiler design and functionality.

2 Introduction to Abstract Syntax Trees (ASTs)

In the realm of compiler construction, the Abstract Syntax Tree (AST) serves as a crucial intermediate representation of the source code. It is a tree representation of the abstract syntactic structure of source code written in a programming language. Unlike the Concrete Syntax Tree (CST), also known as the parse tree, which retains all the details of the grammar's derivation, including non-terminal symbols and syntactic details, the AST focuses solely on the essential semantic structure.

2.1 Key Characteristics of ASTs:

- **Abstraction:** ASTs abstract away syntactic details like parentheses, semicolons, and keywords that are only necessary for parsing but not for semantic analysis or code generation.
- **Semantic Representation:** They represent the program's structure in a way that is closer to its meaning, making it easier for subsequent compiler phases to operate on.
- **Language Independence:** ASTs can be designed to be relatively language-independent, facilitating code optimization and generation for different target platforms.
- **Tree Structure:** They are hierarchical tree structures, reflecting the nested nature of programming constructs.

2.2 Importance of ASTs:

ASTs are fundamental in compilers because they:

- Simplify semantic analysis (e.g., type checking).
- Facilitate code optimization.
- Enable code generation for different target architectures.
- Support program analysis and transformation tools.

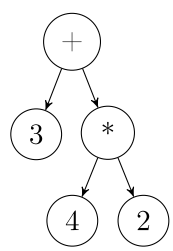
2.3 Examples

2.3.1 Arithmetic Expression

Consider the arithmetic expression: $3 + 4 * 2$.

AST Representation:

The AST for this expression would be:



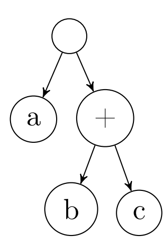
This tree represents the order of operations, where multiplication has higher precedence than addition.

2.3.2 Variable Assignment

Consider the assignment: $a = b + c$.

AST Representation:

The AST for this assignment would be:

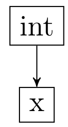


This tree represents the assignment operation and the expression being assigned.

2.3.3 Declaration statements in C

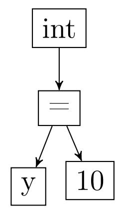
Integer Declaration AST

AST for statement `int x;` is given below:



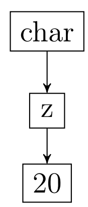
Integer Declaration with Assignment AST

AST for statement `int y=10;` is given below:



Array Declaration AST

AST for statement `char z[20];` is given below:

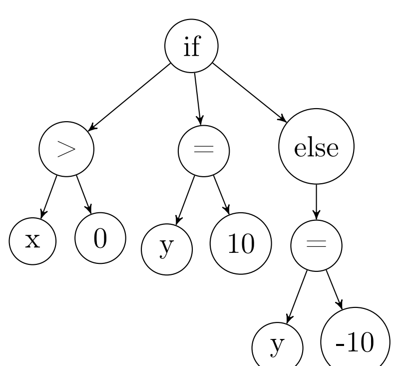


2.3.4 Conditional Statement

Consider the following code snippet:

```
if (x > 0) {
    y = 10;
} else {
    y = -10;
}
```

The AST would represent the conditional structure:

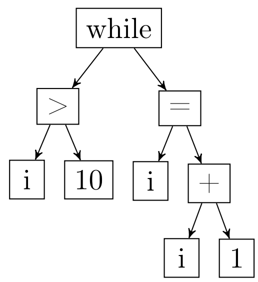


This tree captures the conditional test, the "then" branch, and the "else" branch.

2.3.5 While Loop

```
while (i < 10) {
    i = i + 1;
}
```

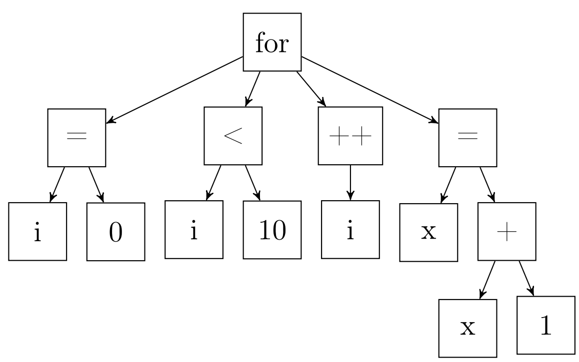
The AST would represent the while structure:



2.3.6 For Loop AST

```
for (i = 0; i < 10; i++) {
    x = x + 1;
}
```

The AST would represent the while structure:



3 Code to generate AST for arithmatic expressions

The following grammar defines arithmetic expressions:

$$
\begin{aligned}
\text{stmt} &\rightarrow \text{expr NL} \\
\text{expr} &\rightarrow \text{expr} + \text{expr} \\
&\quad | \ \text{expr} - \text{expr} \\
&\quad | \ \text{expr} * \text{expr} \\
&\quad | \ \text{expr}/\text{expr} \\
&\quad | \ \text{NUMBER} \\
&\quad | \ (\text{expr})
\end{aligned}
$$

The Bison code is given below:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "ast.h" // Ensure this is included before %union

ASTNode* rootNode = NULL;

extern int yylex();
extern int yyparse();
extern FILE* yyin;
void yyerror(const char* s);
%}
%code requires {
    #include "ast.h" //    Ensures ASTNode is known in parser.tab.h
}
%union {
    int ival;
    ASTNode* ast; // This should work if ast.h is included properly
}


%token NL
%token <ival> NUMBER

%left '+' '-'
%left '*' '/'

%type <ast> expr

%%

stmt : expr NL { printf("Parsing successful!\n");  printAST($1); return 0; }
     ;

expr:  expr '+' expr     { $$ = createOperatorNode(NODE_ADD, $1, $3); }
    | expr '-' expr      { $$ = createOperatorNode(NODE_SUB, $1, $3); }
    | expr '*' expr      { $$ = createOperatorNode(NODE_MUL, $1, $3); }
    | expr '/' expr      { $$ = createOperatorNode(NODE_DIV, $1, $3); }
    | NUMBER             { $$ = createNumberNode($1); }
    | '(' expr ')'       { $$ = $2; }
    ;

%%
```

```c
int main() {
    printf("Enter input string: ");
    yyparse();
    printf("\n");

    return 0;
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
    exit(EXIT_FAILURE);
}
```

The respective flex code is given below:

```
%{
#include "ast.tab.h"
#include "ast.h"
#include <stdio.h>
#include <stdlib.h>
%}

%%

[0-9]+ { yylval.ival = atoi(yytext); return NUMBER; }
"+"    { return '+'; }
"-"    { return '-'; }
"*"    { return '*'; }
"/"    { return '/'; }
"("    { return '('; }
")"    { return ')'; }
[ \t]  { /* Ignore whitespace */ }
.      { printf("Unexpected character: %s\n", yytext); exit(1); }
\n return NL;
%%

int yywrap() {
    return 1;
}
```

The content of `ast.h` is as follows:

```c
#ifndef AST_H
#define AST_H

typedef enum {
    NODE_NUMBER,
    NODE_ADD,
    NODE_SUB,
    NODE_MUL,
    NODE_DIV
} NodeType;

typedef struct ASTNode {
    NodeType type;
    union {
        int value;
        struct {
            struct ASTNode* left;
            struct ASTNode* right;
        } children;
    } data;
} ASTNode;

ASTNode* createNumberNode(int value);
ASTNode* createOperatorNode(NodeType type, ASTNode* left, ASTNode* right);
void printAST(ASTNode* node);
void freeAST(ASTNode* node);

#endif // AST_H
```

The content of `ast.c` is as follows:

```c
#include "ast.h"
#include <stdio.h>
#include <stdlib.h>

// Function to create a number node in the AST
ASTNode* createNumberNode(int value) {
    printf("Creating NUMBER node: %d\n", value);
    ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));
    if (!node) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    node->type = NODE_NUMBER;
    node->data.value = value;
    return node;
}
```

```c
// Function to create an operator node with left and right children
ASTNode* createOperatorNode(NodeType type, ASTNode* left, ASTNode* right) {
    printf("Creating OPERATOR node: %d\n", type);
    ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));
    if (!node) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    node->type = type;
    node->data.children.left = left;
    node->data.children.right = right;
    return node;
}

// Function to print the AST in a readable format
void printAST(ASTNode* node) {
    if (node == NULL) return;

    if (node->type == NODE_NUMBER) {
        printf("%d", node->data.value);
    } else {
        printf("("); // Start expression
        switch (node->type) {
            case NODE_ADD:
                printf("+ ");
                break;
            case NODE_SUB:
                printf("- ");
                break;
            case NODE_MUL:
                printf("* ");
                break;
            case NODE_DIV:
                printf("/ ");
                break;
            default:
                printf("? "); // Handle unknown node types
        }
        printAST(node->data.children.left);
        printf(" ");
        printAST(node->data.children.right);
        printf(")"); // Close expression
    }

}
// Function to free the memory allocated for the AST
void freeAST(ASTNode* node) {
    if (node == NULL) return;
    if (node->type != NODE_NUMBER) {
        freeAST(node->data.children.left);
        freeAST(node->data.children.right);
    }
    free(node);
}
```

Compilation: Assume that the bison code is saved as `ast.y` and the corresponding lex code is saved with `ast.l`. Keep all the files `ast.y, ast.l, ast.c, and ast.h` in the same folder.

- `$ bison -d ast.y`
- `$ flex ast.l`
- `$ gcc ast.tab.c lex.yy.c ast.c -lfl`
- `./a.out`

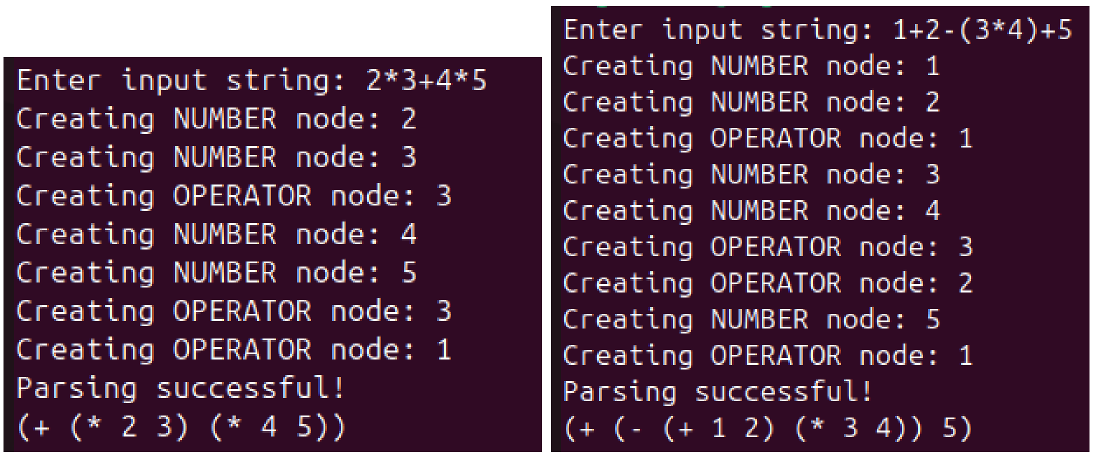


Figure 1: Examples for arithmetic expressions: the final output is given in the form of generalized lists.

Figure 2: The tree structures for the AST (given in the list form in the Figure 1.

One can use the following code to generate the above tree structures:

```
from nltk.tree import *

# Corrected tree representation with explicit binary structure
text = "(+ (- (+ 1 2) (* 3 4)) (5))"

tree = Tree.fromstring(text)
tree.pretty_print(unicodelines=True, nodedist=5)
```

4 Exercises

1. Extend the function to handle assignment statements in the general form. Example: $a = b + 10 * c$.

2. Extend the function to handle declaration statements in C. Examples:

   - `int x;`
   - `int x, y=10, z;`
   - `int x, y=10, z=y+5;`

3. Extend the functions to handle multiple statements in the problem. For example

   ```c
   int x=10, y=5;
   x=x+5;
   y=y+x*5/8;
   x=y*y;
   ```

   Read the input from a text file.

   You decide the structure of the AST.

4. (Homework) Extend the function to handle conditional statements, both simple if and if-else. Assume that the true/false block contains multiple statements.

5. (Homework) Extend the function to handle looping statements: while and for. Assume that the body of the loop contains many statements.