

Labs #4

Semantic analysis phase of a compiler

AST construction - tree representation of entire program

AST printing - visualizes tree structure

Symbol Table - tracking variable declarations & scopes

Semantic checking - validating type rules, var usage, etc

Source code \rightarrow Scanner (Flex) \rightarrow Parser (Bison) \rightarrow AST \rightarrow Semantic checker

Code fun

AST Structure

AST node needs to represent what?

Build tree as parser is parsing

Parser.y

expression \rightarrow expression + expression

declaration \rightarrow type ID = expression

statement \rightarrow IF (expression) statement ELSE statement

Each needs to create an AST node \rightarrow ast.h

Discriminated Union Pattern

not - operator, left child, right child - binary expression

-x - unary expression - operator, one child

42 - literal - just val

vec3(b2,3) - constructor - need type + obj list

#(cond) { ... } - need cond, then branch, else branch

Each AST node will need diff type of data

↳ Discriminated Union

struct Node {
 node_type kind; // Discriminant / which variant is active
 union {
 use one per node
 };

struct { ... } binary_expr;
struct { ... } unary_expr;
struct { ... } Literal;

{:
};

Check kind, then access corresponding union member

#(cond + kind == BINARY_EXPRESSION_NODE)
// always

{

Bison Semantics

Parses builds parse tree bottom up

Bison - attach code for each grammar rule when that rule is reduced

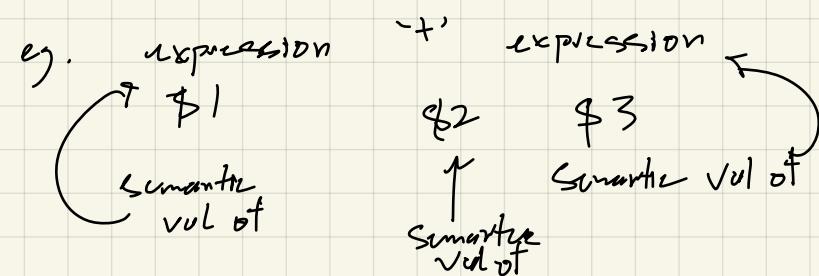
expression

: expression '+' expression

{ y TRACE(...); }

\$\$ \$1 \$2 \$3

Return semantic values of symbols in the rule



\$\$ semantic val being created

% union {

```
int as_int;
float as_float;
char * as_str;
node * as_ast;
```

}

Building AST for any eg.

expression

: expression '+' expression

{ \$1 \$3 already AST nodes built bottom up}

\$\$ = ast_allocate(BINARY_EXPRESSION_NODE, '+', \$1, \$3);

{

Bottom up parsing

children parsed before parents

Symbol Tables

Dictionary that maps var names to info about var

what vars exist at each point in program

what types they have (int, float)

Are they const? → cannot be reassigned

Current scope (shadowing variables)

Cactus Stack approach

Global scope - { x: int y: float }

Inner scope - { x: bool z: int } ← shadow global

Inner most - { w: vec3 }

Looks up a var

↳ check innermost scope

If not found, check parent

Repeat until found / reached global scope

When entering a new scope => { => push new table

Exit scope => pop table

Expanding AST nodes

Need nodes for

Expressions:

Binary operators

Unary operators

Literals int float bool

Variables

Function calls

Constructors

Statements

Assignment

If/else

Scopes {}

Declarations

Variable declarations

w/ without const

w/ without initializer

Struct node - {

node kind kind;

Union

// exp

struct {

int op;

Node left;

Node right;

} binary_expr;

struct {

int op;

} unary_expr;

// statements

// statement types

}

Node Structure

Expressions - binary ops, literals, var, func calls, constructors

Statements - assignments if/else scopes, empty statements

Declarations - type, identifier, optional const, optional initializer

Type s: int, bool, float, vec2/3/4,ivec2/3/4 base 2/3/4

Bison:

```
%{
    // Code
    #include "ast.h"
%}
```

// Declarations

```
%union {
```

tokens ...

%type -> fill out

%left ...

%start program

%%

// Grammar

program

: scope

{ ast = \$1; }

;

Scope ,;

What info does grammar rule capture

How do I represent it in my AST?

Purpose

What info need to preserve - type, value?, etc?

g. TYPE_NODE

type code, user data

Scanner

vec2(3)) { -- }

value in \$1

g. type

: INT-7

| FVEC4T

Parser.y

to type declarations - ^{Left Bound} non terminals produce node*

Stores as ast from field

Program rules

ast_allocate, ast_free, ast_print

Set up already by parser.y scanner.h

Structs

symbol_entry

name
type
vec size
const
r
w

Symbol table

Dark pointer to entries
size + count
parent

Functionality

Simple hash func

Create symbol table

Free symbol table

Insert

Looking local scope

Lookup in all scopes

Traversing AST

Enter scope node - new symbol table, push onto stack

For every declaration node - insert into current table

See Var node - lookup into current table

Exit scope - pop symbol table from stack

Traverse AST recursively maintaining current scope pointer