

Lab #4

Semantic analysis phase of a compiler

AST construction - tree representation of entire program

AST printing - Visualize 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 Gen

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

op_g - operator, left child, right child - binary expression

u₁ - unary expression - operator, one child

u₂ - literal - just need val

vec3(1,2,3) - constructor need type + arg list

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

Each AST node will need diff type of data

\hookrightarrow Discriminated Union

```
struct node {
    node *left; kind;  $\Rightarrow$  Discriminant / which variant is active
    union {
        struct { ... } binary_expr;
        struct { ... } unary_expr;
        struct { ... } literal;
    };
};
```

Use one per node

Check kind, then access corresponding union member

```
#if (node & kind == BINARY_EXPRESSION_NODE) {
    // okay
}
```

Bison Semantics

Parsing builds parse tree bottom up

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

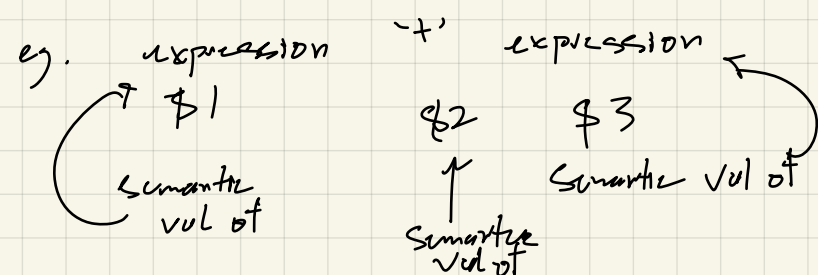
expression

: expression '+' expression

{ y TRACE(...); }

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

Reference semantic values of symbols in the rule



% union {

int as_int;

float as_float;

char * as_str;

void * as_ptr;

}

Building AST for stmt y.

expression

: expression '+' expression

{ \$1 \$3 already AST nodes (built bottom up)

\$\$ = ast_allocate (BINARY_EXPRESSION_NODE, '2', \$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)

Linked lists approach

Global Scope — { x: int y: float }

Inner Scope — { x: bool z: int } ← shadow global x

Inner most — { w: vec3 }

Look up a var

↳ Check innermost scope

If not found, check parent

Repeat until found / reached global scope

When entering a new scope ⇒ { = x push new table

Exit scope ⇒ pop table

Expanding AST nodes

Need nodes for

Expressions:

Binary operators

Unary operators

Literals int float bool

Variables

Func 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, lace 2/3/4

Bison:

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

// Declarations

%union { }

% token ...

% type -> fill out

% left ...

% start program

% %

// Grammar

program

{ scope

{ ast = ? ; }

;

scope , ... }

what info does grammar rule capture

how do I represent it in my AST

purpose

what info need to preserve - type, values?, size?

eg. TYPE_NODE

type code, version

eg. type

: INT_1

| IVECT

Scanner

vec2/3/4)

{ ... }

value in #1

Parser.y
%< type declarations - non terminals produce node*
store as_ast union field
program rules

ast_allocate, ast_free, ast_print
Set up already by parser.y, scanner.l

Structs

symbol_entry

name
type
use size
const
v
w

Symbol table

Double pointer to entries
Size + count
parent

Functionality

Simple hash func

Create symbol table

Free symbol table

Insert

Lookup 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