Lab #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 → Scanner (Flex) → Parser (Bison) → AST → Semantic checker

Code Gen

AST Structure

AST node needs to represent what?

Build tree as parser is parsing

Parser.y

```
expression → expression + expression
declaration → type ID = expression
statement → IF ( expression ) statement ELSE statement
```

Each needs to create an AST node → ast.h

Discriminated Union Pattern

- a+b — operator, left child, right child — binary expression
- -x — unary expression — operator, one child
- 42 — 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

↳ Discriminated Union

```
struct node{
    node_kind 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

```
if (node->kind == BINARY_EXPRESSION_NODE){
    // access
}
```

Bison Semantics

Parsing builds parse tree bottom up

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

```
expression
  : expression '+' expression
      { yTRACE(...) }
```

$$ $1 $2 $3

Reference semantic values of symbols in the rule

e.g. expression '+' expression
$1 $2 $3
semantic val of
semantic val of
Semantic val of

$$ semantic val being created

```
%union {
    int as_int;
    float as_float;
    char * as_str;
    node *as_ast;
} ...
```

Building AST for stg 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 Stacks 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 => { => 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

Types: int, bool, float, vec2/3/4, ivec2/3/4 bvec 2/3/4

Bison:

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

// Declarations
%union { }
% token ...

%type -> fill out
% left   ...
%start program

%%
// Grammar

program
    : scope
      { ast = $1;}
    ;

scope ;...
```

scanner.l

```
 vec(2|3|4)    { ... }
  value in $1
```

what info does grammar rule capture

How do I represent it in my AST

e.g. type

```
  : INT_T
  | IVEC_T
```

Purpose

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

e.g. TYPE_NODE

type_code, vec_size

Parser.y

% type declarations — non terminals produce node*

tell Bison

stored 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

vec size

const

r

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