Proyecto Final

Teoría de Lenguajes y Programación

Osornio López Daniel Alejandro 0244685@up.edu.mx

Barba Pérez Jorge 0237328@up.edu.mx

2023-11-27

Flex

- Movimos distintos elementos de la gramática al analizador léxico.
- El trabajo de reconocer caracteres individuales es del lexer.
- Devolvemos `struct Symbol` desde flex.

```
{SIGNO}{ENTERO} {
  nchar += yyleng;
  yylval.snum = atoll(yytext);
  return CONST_ENTERA;
}
```

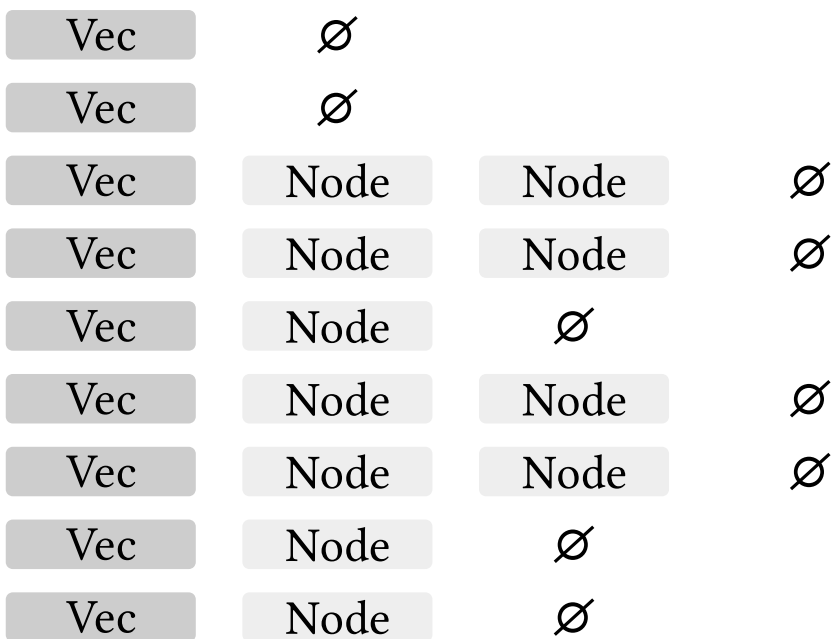
Bison

- Permitir múltiples declaraciones de variables y constantes
- `write` y `writeln` con una lista de expresiones + cadenas constantes

```
lectura_instruccion : KW_READ '(' IDENT ')' {
  assert_sym_exists(&$3);
  tree_init(&$$, sizeof(Node));
  Node * n = ast_create_node(&$$, NRead, Void);
  n->value.read = (ReadNode){
      .newline = 0, .target_symbol = $3,
  };
};
```

HashSet

- Conformado por un vector general (gris obscuro), donde cada valor del vector es otro vector de colisiones
- Se utiliza una función de digestión que consume el nombre y ámbito
- Usamos una estructura con unión anotada



Symbol

- La tabla de símbolos contiene solo valores de este tipo.
- Pre-llenamos los símbolos en el lexer, completamos en bison.
- Con la variante de `SymbolType` sabemos cómo acceder a la unión `SymbolInfo`
- Todos los símbolos tienen un tipo de dato asociado.

```
struct Symbol {
        StrSlice name;
        SymbolType type;
        SymbolInfo info;
        DataType asoc_type;
        size_t scope;
        size_t line;
        size_t nchar;
        Vec refs;
};
```

Validar símbolos

```
lectura_instruccion : READ '('
IDENT ')' {
    assert_sym_exists(&$3);
    /* ... */
};

decl_const : decl CONST IDENT '='
ENTERO ';' {
    assert_not_sym_exists(&$3);
    hashset_insert(&tabla, &$3);
};
```

- `assert_sym_exists` y `assert_non_sym_exist` para verificar la existencia/insertar símbolos
- Mensaje de error para el usuario
- Completamos información en bison

Tree

```
struct Tree {
    Vec relations;
    Vec values;
};

subprogramas : subprogramas
subprograma_declaracion ';' {
    $$ = $1;
    tree_extend(&$$, &$2, 0, 0);
}
```

- Usamos una lista de adyacencia para describir relaciones
- Los nodos se almacenan en un vector
- La mayoría de no terminales tienen como valor asociado un árbol
- Contruimos el árbol de abajo hacia arriba

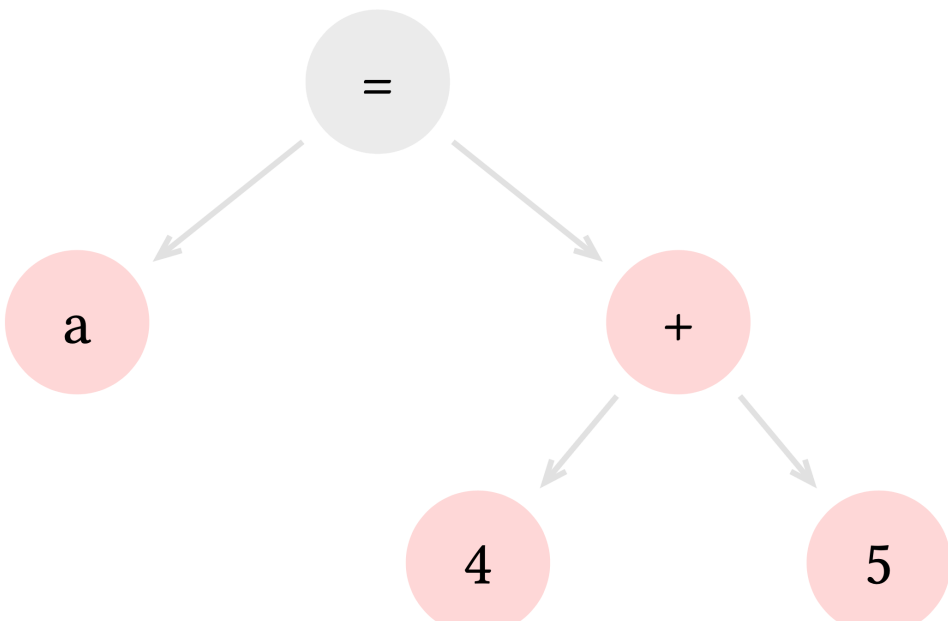
Node

- Todos los elementos del AST son del tipo `Node`
- Utiliza un enumerador anotado para almacenar infromación específica
- Cada nodo sabe como imprimirse a sí misno a un *stream*
- Cada nodo tiene un tipo de dato asociado

```
struct Node {
    NodeType node_type;
    NodeValue value;
    size_t id;
    DataTypeE asoc_type;
};
```

AST

- Los árboles están anotados con un tipo asociado
- Las operaciones entre árboles solo están permitidas si cumplen condiciones
- Las operaciones pueden cambiar el valor asociado (p. ej. la operación `int && int == bool`)



Verificando árboles

- Las verificaciones en expresiones, asignaciones a variables, argumentos, etc.
- Si no se cumplen los requisitos se muestra un mensaje de error.
- Si no se presenttan errores se realiza la unión y se asigna el tipo asociado que corresponsa

```
r_expr : r_expr RELOP_OR r_and {
  tree_init(&$$, sizeof(Node));

  Node * lhs = ast_get_root(&$1);
  Node * rhs = ast_get_root(&$3);
  assert_expr_type(lhs, rhs);
  /* ... */
  tree_extend(&$$, &$1, 0, 0);
  tree_extend(&$$, &$3, 0, 0);
}
```

Codegen

```
if (err == 0) node_display_id(0, OUT_FILE, &ast, &tabla, 0);
```

- Si no se encontraron errores, se genera el código.
- Cada nodo sabe cómo mostrarse, y en esos pasos mostrará sus hijos.
- Recursivamene se va imprimiendo todo el código

```
case NIf: {
  mostrar(f, "if ("); mostrar_nodos(f, hijos[0]);
  mostrar(f, ") {");
  mostrar_nodos(f, hijos[1]); mostrar(f, "}\n");
}
```

Gracias!