Semantic Analysis

Mini-C Compiler

CS4031 - Compiler Construction | Fall 2025

Team Members

Subhan (22K-4316)

Sadiq (22K-4303)

Muhammad Ahmed Haque (22K-4232)

Muhammad Irtiza (22K-4638)

Umer (22K-4160)

Junaid (22K-4369)

December 2025

1. Overview

Semantic analysis is the third phase of the Mini-C compiler. It performs:

- Variable Resolution - Binding identifiers to their declarations
- Type Checking - Ensuring type compatibility in expressions
- Symbol Table Construction - Managing scopes and declarations

Location: src/backend/typechecker/

2. Symbol Table Structure

2.1 Symbol Table Entry

```
class SymbolEntry:
    name: str           # Identifier name
    type: Type          # Data type (Int, Pointer, Array, etc.)
    storage: Storage    # Storage class (Local, Static, Extern)
    scope_level: int    # Nesting depth
    offset: int         # Stack offset (for locals)
    is_defined: bool    # Whether fully defined
```

2.2 Storage Classes

Storage Class	Description	Example
Local	Stack-allocated local variable	int x = 5;
Static	Persists across calls	static int count = 0;
Extern	Defined externally	extern int global;
Global	File-scope variable	Top-level int g;

2.3 Symbol Table Operations

Operation	Description
insert(name, entry)	Add new symbol to current scope
lookup(name)	Search for symbol in all scopes
lookup_current(name)	Search only in current scope
enter_scope()	Create new nested scope
exit_scope()	Remove current scope

3. Type System

3.1 Type Hierarchy

```
Type
+-- Void
+-- Arithmetic Types
|    +-- Integer Types
|    |    +-- Char, SChar, UChar
|    |    +-- Int, UInt
|    |    +-- Long, ULong
|    +-- Floating Types
|         +-- Double
+-- Derived Types
|    +-- Pointer(referenced: Type)
|    +-- Array(element: Type, size: int)
|    +-- Structure(tag: str)
+-- Function Types
     +-- FunType(params: List[Type], return_type: Type)
```

3.2 Type Sizes and Alignment

Type	Size (bytes)	Alignment
char	1	1
int	4	4
long	8	8
double	8	8
pointer	8	8
array[N]	N * elem_size	elem_align
struct	sum of members	max member align

4. Semantic Rules

4.1 Declaration Rules

Rule	Description	Violation Example
D1	Variables must be declared before use	x = 5; (x not declared)
D2	No duplicate declarations in scope	int x; int x;
D3	Function params must be unique	int f(int a, int a)
D4	Arrays must have positive size	int arr[-1];
D5	Struct members must be unique	struct { int x; int x; }

4.2 Type Checking Rules

Rule	Description	Violation Example
T1	Arithmetic ops require numeric operands	"hello" + 5
T2	Comparison requires compatible types	5 < "str"
T3	Logical ops require scalar types	struct_var && 1
T4	Assignment requires compatible types	int *p = 5;
T5	Function args must match params	f(1,2) when f(int)
T6	Return type must match declaration	int f() { return "x"; }
T7	Array subscript must be integer	arr[3.14]
T8	Dereference requires pointer type	*42

5. Symbol Table Example

Source Code:

```
int global_var = 10;

int factorial(int n) {
    if (n <= 1) {
        return 1;
    }
    int result = n * factorial(n - 1);
    return result;
}

int main(void) {
    int x = 5;
    int y = factorial(x);
    return y;
}
```

Symbol Table - Global Scope (Level 0):

Name	Type	Storage	Defined
global_var	Int	Global	Yes
factorial	FunType(Int->Int)	Global	Yes
main	FunType(Void->Int)	Global	Yes

Symbol Table - Inside factorial (Level 1):

Name	Type	Storage	Offset
n	Int	Local	-8
result	Int	Local	-12

Symbol Table - Inside main (Level 1):

Name	Type	Storage	Offset
x	Int	Local	-8
y	Int	Local	-12

6. Integer Promotions and Conversions

6.1 Integer Promotion Rules

```
def promote_integer(type):
    """Promote small integers to int"""
    if type in [Char, SChar, UChar]:
        return Int
    return type
```

6.2 Usual Arithmetic Conversions

```
def common_type(t1, t2):
    """Find common type for binary operation"""
    if t1 == t2:
        return t1
    if t1 == Double or t2 == Double:
        return Double
    if t1 == ULong or t2 == ULong:
        return ULong
    if t1 == Long or t2 == Long:
        return Long
    if t1 == UInt or t2 == UInt:
        return UInt
    return Int
```

7. Type Checking Examples

Valid Type Usage:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;           // Array decays to pointer - OK
int val = ptr[2];         // Pointer subscript - OK
int sum = arr[0] + val;   // Int + Int - OK
```

Type Errors:

```
int x = "hello";          // ERROR: Cannot assign char* to int
double *dp = &x;          // ERROR: Cannot assign int* to double*
int y = arr;              // ERROR: Cannot assign array to int
x();                      // ERROR: x is not a function
```