Semantic Analysis

Uday Khedker

(www.cse.iitb.ac.in/˜uday)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

Feb 2025

Outline

- The role of semantic analysis:

  The need of semantic validation, examples of errors
- The basic concepts for semantic analysis
- Applications of semantic analysis
  - IR generation
  - Name and scope analysis
  - Declaration processing
  - Type analysis
- Run time support
  - Activation records
  - Stack, static, and heap allocation,
  - Function prologue, making a call, returning a call, function epilogue

Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

The Role of Semantic Analysis

The Role of Semantic Analysis

- Establishing semantic validity of programs
  - What kinds of errors are possible in a program?
  - What kind of analysis can check these errors?
- Generating intermediate code (AST or Three-address code)
- Generating code for run time support (procedure calls and returns)

Why Separate Semantic Analysis from Syntax Analysis?

The constraints defining semantic validity cannot be described by context free grammars

Why Separate Semantic Analysis from Syntax Analysis?

The constraints defining semantic validity cannot be described by context free grammars

- The constraint "declare a variable before its use" can be described by a language $\{wcw \mid w \in \Sigma^*\}$ where w is the lexeme of a variable

  (the lexeme appearing in a use must match the lexeme appearing in the corresponding declaration)

Why Separate Semantic Analysis from Syntax Analysis?

The constraints defining semantic validity cannot be described by context free grammars

- The constraint "declare a variable before its use" can be described by a language $\{wcw \mid w \in \Sigma^*\}$ where w is the lexeme of a variable

  (the lexeme appearing in a use must match the lexeme appearing in the corresponding declaration)

- The constraint "the number of actual parameters in a call must match the number of formal parameters of the procedure" for a program with two procedures can be described by a language $\{fa^n gb^m fc^n gd^m \mid n \geq 1, m \geq 1\}$ where
  - the formal parameters of procedure f are represented by a string of a's and its actual parameters are represented by a string of c's, and
  - the formal parameters of procedure g are represented by a string of b's and its actual parameters are represented by a string of d's

Why Separate Semantic Analysis from Syntax Analysis?

The constraints defining semantic validity cannot be described by context free grammars

- The constraint "declare a variable before its use" can be described by a language $\{wcw \mid w \in \Sigma^*\}$ where w is the lexeme of a variable

  (the lexeme appearing in a use must match the lexeme appearing in the corresponding declaration)

- The constraint "the number of actual parameters in a call must match the number of formal parameters of the procedure" for a program with two procedures can be described by a language $\{fa^n gb^m fc^n gd^m \mid n \geq 1, m \geq 1\}$ where
  - the formal parameters of procedure f are represented by a string of a's and its actual parameters are represented by a string of c's, and
  - the formal parameters of procedure g are represented by a string of b's and its actual parameters are represented by a string of d's

These languages are not context free and hence cannot be described by context free grammars

So How Do We Perform Semantic Analysis?

- Using context sensitive grammars for parsing is expensive
- Practical compilers use context free grammars to admit a superset of valid sentences and prune out invalid sentences by imposing context sensitive restrictions

Context-Sensitive Grammar (CSG)
• A type of formal grammar where production rules are of the form alpha A beta -> alpha gamma beta, where A is a non-terminal and alpha, beta, and gamma are strings (gamma != epsilon).
• Generates Context-Sensitive Languages (CSL), which are recognized by Linear Bounded Automata (LBA).
• More powerful than Context-Free Grammar (CFG) but less expressive than Recursively Enumerable Languages.
• Used in natural language processing (NLP) and some programming languages, but parsing is computationally expensive (PSPACE-complete).

So How Do We Perform Semantic Analysis?

- Using context sensitive grammars for parsing is expensive
- Practical compilers use context free grammars to admit a superset of valid sentences and prune out invalid sentences by imposing context sensitive restrictions
  - For recognizing language $\{wcw \mid w \in \Sigma^*\}$,
    - admit all sentences in $\{xcy \mid x, y \in \Sigma^*\}$,
    - enter x in a symbol table during declaration processing, and
    - when uses are processed, lookup the symbol table and check if $y = x$

Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

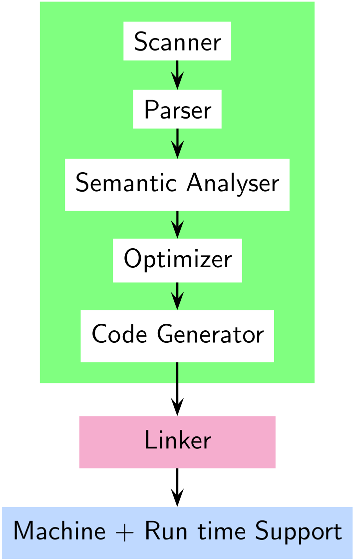
So How Do We Perform Semantic Analysis?

- Using context sensitive grammars for parsing is expensive
- Practical compilers use context free grammars to admit a superset of valid sentences and prune out invalid sentences by imposing context sensitive restrictions
  - For recognizing language $\{wcw \mid w \in \Sigma^*\}$,
    - admit all sentences in $\{xcy \mid x, y \in \Sigma^*\}$,
    - enter x in a symbol table during declaration processing, and
    - when uses are processed, lookup the symbol table and check if $y = x$
  - For language $\{fa^n gb^m fc^n gd^m \mid n \geq 1, m \geq 1\}$,
    - admit all sentences in $\{fa^n gb^m fc^i gd^j \mid n \geq 1, m \geq 1, i \geq 1, j \geq 1\}$,
    - enter a^n and b^m as attributes of procedures f and g in a symbol table when function declarations/definitions are processed,
    - match c^i with a^n when a call to f is encountered, and
    - match d^j with b^m when a call to g is encountered

So How Do We Perform Semantic Analysis?

- Using context sensitive grammars for parsing is expensive
- Practical compilers use context free grammars to admit a superset of valid sentences and prune out invalid sentences by imposing context sensitive restrictions
  - For recognizing language $\{wcw \mid w \in \Sigma^*\}$,
    - admit all sentences in $\{xcy \mid x, y \in \Sigma^*\}$,
    - enter x in a symbol table during declaration processing, and
    - when uses are processed, lookup the symbol table and check if $y = x$
  - For language $\{fa^n gb^m fc^n gd^m \mid n \geq 1, m \geq 1\}$,
    - admit all sentences in $\{fa^n gb^m fc^i gd^j \mid n \geq 1, m \geq 1, i \geq 1, j \geq 1\}$,
    - enter a^n and b^m as attributes of procedures f and g in a symbol table when function declarations/definitions are processed,
    - match c^i with a^n when a call to f is encountered, and
    - match d^j with b^m when a call to g is encountered
  - The general strategy is to define and compute some attributes of the symbols of a context free grammar and communicate the semantic information between them through the attributes

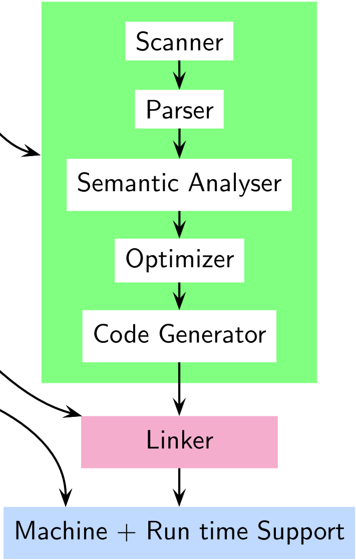
Ensuring Validity of Programs = Detecting and Prohibiting Errors



Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

Ensuring Validity of Programs = Detecting and Prohibiting Errors

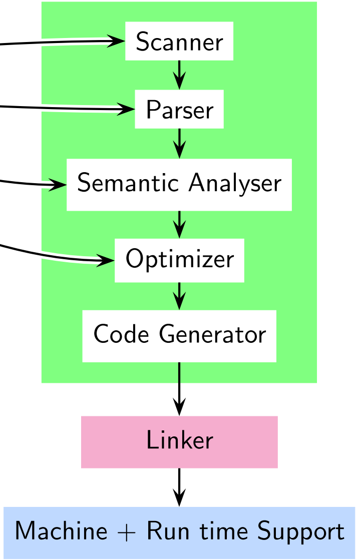
- Compile time errors. Compilation fails
- Link time errors. Linking fails
- Run time errors. Execution fails



Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

Ensuring Validity of Programs = Detecting and Prohibiting Errors

- Compile time errors. Compilation fails
  - Lexical error
  - Syntax error
  - Semantic error
- Link time errors. Linking fails
- Run time errors. Execution fails

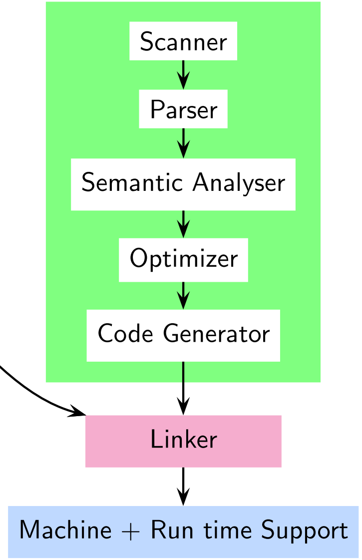


Ensuring Validity of Programs = Detecting and Prohibiting Errors

- Compile time errors. Compilation fails
  - Lexical error
  - Syntax error
  - Semantic error

- Link time errors. Linking fails
  Missing functions, global variables
  ("undefined reference to vtable for f")

- Run time errors. Execution fails

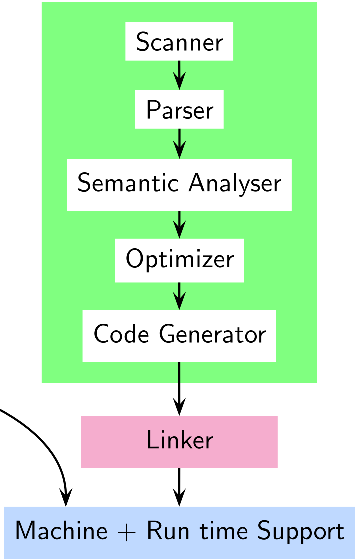


Ensuring Validity of Programs = Detecting and Prohibiting Errors

- Compile time errors. Compilation fails
  - Lexical error
  - Syntax error
  - Semantic error
- Link time errors. Linking fails

  Missing functions, global variables

  ("undefined reference to vtable for f")
- Run time errors. Execution fails
  - Logical error. Execution completes but gives wrong result
  - Undefined behaviour. Execution either aborts or gives wrong result



Undefined Behaviour, Exceptions, and Unspecified Behaviour

- Undefined behaviour. Unchecked prohibited behaviour flagged by the language
  - No responsibility of the compiler or its run time support
  - May have unpredictable outcomes
    The execution may abort or give unexpected result
  - A compiler is legally free to do anything
    Including formatting your disk or launching a missile ;-)

Accessing an out-of-bounds array element -> int x = arr[1000]; // UB if arr has fewer elements

- Unspecified behaviour (aka implementation-defined behaviour)
  - Valid feature whose implementation is left to the compiler
  - The available choices do not affect the result but may influence efficiency
  - Examples. The order of evaluation of arguments to a function call, or subexpressions

Function arguments evaluation order -> printf("%d %d", i++, i++); // Unspecified order of evaluation

- Exceptions. Prohibited behaviour checked by the run time support

Division by zero -> int x = 10 / 0; // Throws runtime exception

Undefined Behaviour, Exceptions, and Unspecified Behaviour

- Undefined behaviour. Unchecked prohibited behaviour flagged by the language
  - No responsibility of the compiler or its run time support
  - May have unpredictable outcomes
    The execution may abort or give unexpected result
  - A compiler is legally free to do anything
    Including formatting your disk or launching a missile ;-)

Practical compilers try to detect them and issue warnings (and not errors)

- Unspecified behaviour (aka implementation-defined behaviour)
  - Valid feature whose implementation is left to the compiler
  - The available choices do not affect the result but may influence efficiency
  - Examples. The order of evaluation of arguments to a function call, or subexpressions

Practical compilers make choices based on well defined criteria

- Exceptions. Prohibited behaviour checked by the run time support

Practical compilers try to detect these at compile time

Undefined Behaviour, Exceptions, and Unspecified Behaviour

- Undefined behaviour. Unchecked prohibited behaviour flagged by the language
  - No responsibility of the compiler or its run time support
  - May have unpredictable outcomes
    The execution may abort or give unexpected result
  - A compiler is legally free to do anything

Java follows the exception approach for predictability, whereas C/C++ follows the undefined behaviour approach for efficiency

predictability means that the compiler must document how it handles the behavior, ensuring that it is consistent for that specific implementation. While different compilers may choose different implementations, the behavior remains deterministic and predictable within the same compiler and settings.

subexpressions

Practical compilers make choices based on well defined criteria

- Exceptions. Prohibited behaviour checked by the run time support

Practical compilers try to detect these at compile time

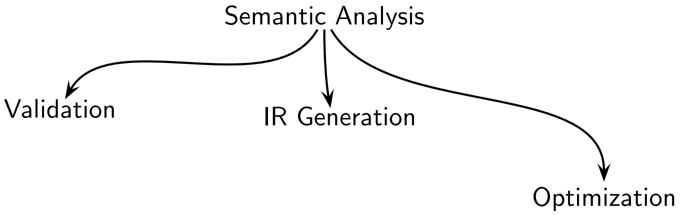
Examples of Undefined Behaviour in C

- Memory violations
  - Dereferencing a NULL pointer
  - Out-of-bounds array access
  - Modifying a string literal
  - Accessing uninitialized variables
  - Invalid pointer arithmetic
  - Using a pointer after free (dangling pointer)
  - Accessing local variables after function return
- Compute violations
  - Division by zero
  - Signed integer overflow
  - Overflow or underflow in floating-point operations
  - Failing to return a value from a non-void function
  - Infinite recursion without a base case

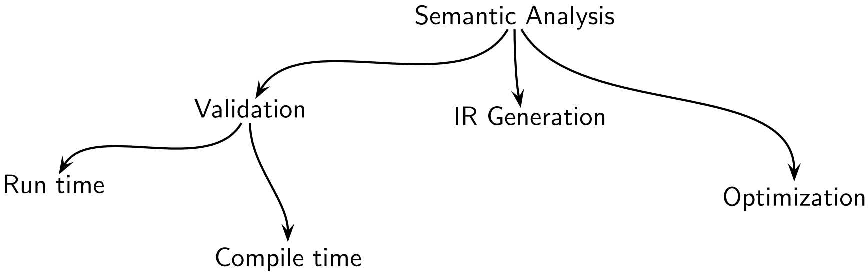
Examples of Unspecified Behaviour in C

- Order of evaluation of function arguments
- Order of evaluation of subexpressions in an expression
- Overflow or underflow for unsigned integers
- Alignment of structures and unions
- Memory layout of `struct` and `union` types
- Padding added to structures

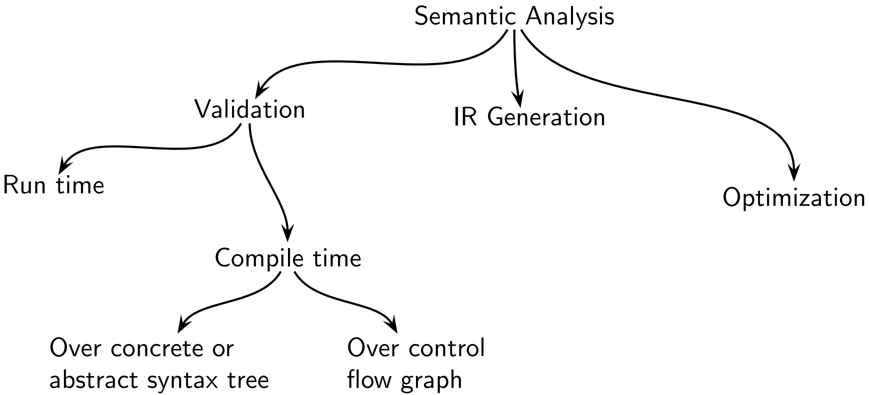
Different Forms of Semantic Analysis



Different Forms of Semantic Analysis



Different Forms of Semantic Analysis



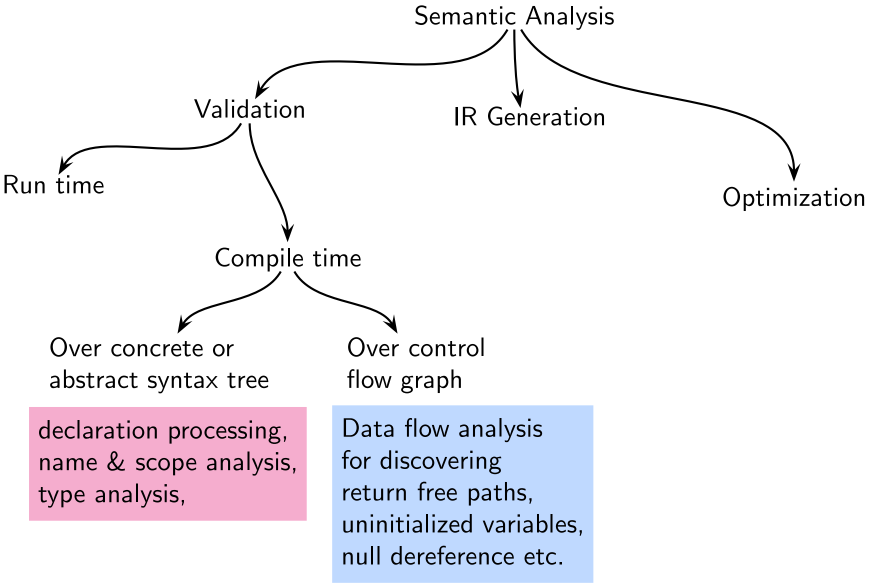
Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

Different Forms of Semantic Analysis

- Semantic Analysis
  - Validation
    - Run time
    - Compile time
      - Over concrete or abstract syntax tree
      - Over control flow graph
  - IR Generation
  - Optimization

declaration processing,
name & scope analysis,
type analysis,

Different Forms of Semantic Analysis



Different Forms of Semantic Analysis

- Semantic Analysis
  - Validation
    - Run time
      - dynamic typing, exception handling
    - Compile time
      - Over concrete or abstract syntax tree
        - declaration processing, name & scope analysis, type analysis,
      - Over control flow graph
        - Data flow analysis for discovering return free paths, uninitialized variables, null dereference etc.
  - IR Generation
  - Optimization

Different Forms of Semantic Analysis

- Semantic Analysis
  - Validation
    - Run time
      - dynamic typing, exception handling
    - Compile time
      - Over concrete or abstract syntax tree
        - declaration processing, name & scope analysis, type analysis,
      - Over control flow graph
        - Data flow analysis for discovering return free paths, uninitialized variables, null dereference etc.
  - IR Generation
    - abstract syntax tree or three-address code, code for handling procedure calls
  - Optimization

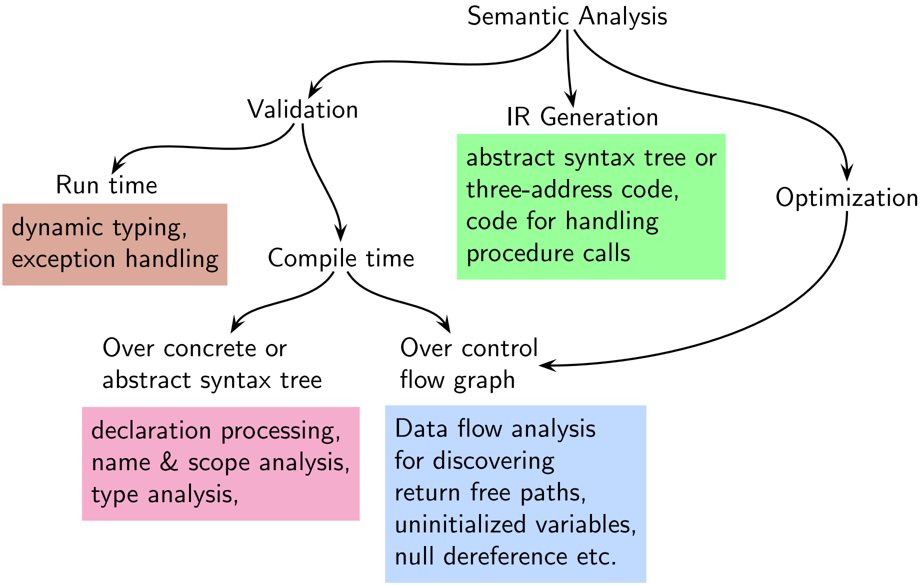
Overcontrol Flow Graph: A control flow graph (CFG) with extra edges or nodes, representing redundant or unnecessary control paths in a program. Example: A loop with unreachable branches or excessive condition checks.

Data Flow Analysis: A technique to track how data moves through a program, used for detecting return-free paths, uninitialized variables, and null dereferences.

Return-Free Path Detection: Ensures all code paths eventually return in functions that must return a value.

Null Dereference Detection: Identifies cases where a null pointer might be accessed.

Different Forms of Semantic Analysis



Different Forms of Semantic Analysis

Semantic Analysis

- Validation
  - Run time
    - dynamic typing, exception handling
  - Compile time
    - Over concrete or abstract syntax tree
      - declaration processing, name & scope analysis, type analysis,
    - Over control flow graph
      - Data flow analysis for discovering return free paths, uninitialized variables, null dereference etc.
- IR Generation
  - abstract syntax tree or three-address code, code for handling procedure calls
- Optimization

We will cover these

We will cover the basics of these

How Can a Compiler Ensure Run Time Validation?

- Assume that a compiler decides to guard against null pointer dereference
- Every occurrence of $*x$ can be replaced by a code that has the effect of the following expression

$$(x != NULL)\,?\, *x : complain()$$

where function *complain* is a part of the run time support created by the compiler

- This is not a source level change but the IR of the program would be instrumented
- Note that this overhead slows down the program execution

The Role of Semantic Analysis

Examples of Errors

Syntax Directed Definitions

Generating IR

Syntax Directed Translation Schemes

Type Analysis

Name and Scope Analysis

Declaration Processing

Examples of Errors

Acknowledgements

This section is based entirely on the material developed by Prof. Biswas

Observations About Program p0.c

```
using namespace std;

#include <iostream>
/*
 * Test Program
 *
int main()
{ int a = b;
  int b = 5;
  return 0;
}
```

Observations About Program p0.c

```
using namespace std;

#include <iostream>
/*
 * Test Program
 *
int main()
{ int a = b;
  int b = 5;
  return 0;
}
```

- Unterminated comment

Observations About Program p0.c

```
using namespace std;

#include <iostream>
/*
 * Test Program
 *
int main()
{ int a = b;
  int b = 5;
  return 0;
}
```

- Unterminated comment
- Lexical error

Observations About Program p1.c

```
using namespace std;

#include <iostream>

int main()
{ int a = b;
  int b = 5;
  return 0;
}
```

Observations About Program p1.c

```
using namespace std;

#include <iostream>

int main()
{ int a = b;
  int b = 5;
  return 0;
}
```

- Declaration of b appears after its definition

Observations About Program p1.c

```
using namespace std;

#include <iostream>

int main()
{ int a = b;
  int b = 5;
  return 0;
}
```

- Declaration of b appears after its definition
- Cannot be identified by the scanner
- Cannot be identified by the parser
  - Our grammar is context-free
  - This needs recording and examining context
  - A variable is used in the context of its declaration

Observations About Program p1.c

```
using namespace std;

#include <iostream>

int main()
{ int a = b;
  int b = 5;
  return 0;
}
```

- Declaration of b appears after its definition
- Cannot be identified by the scanner
- Cannot be identified by the parser
  - Our grammar is context-free
  - This needs recording and examining context
  - A variable is used in the context of its declaration
- Semantic error (name and scope analysis)

Observations About Program p2.c

```
using namespace std;

#include <iostream>

int main()
{ int a = b, b = 5;
  return 0;
}
```

Observations About Program p2.c

```
using namespace std;

#include <iostream>

int main()
{ int a = b, b = 5;
  return 0;
}
```

- Declaration of b appears after its use even if it is within the same declaration statement

Observations About Program p2.c

```
using namespace std;

#include <iostream>

int main()
{ int a = b, b = 5;
  return 0;
}
```

- Declaration of b appears after its use even if it is within the same declaration statement
- Semantic error (name and scope analysis)

Observations About Program p3.c

```
using namespace std;

#include <iostream>

int main()
{ float b;
  int b = 5;
  return 0;
}
```

Observations About Program p3.c

```
using namespace std;

#include <iostream>

int main()
{ float b;
  int b = 5;
  return 0;
}
```

- Redeclaration of b with different types

Observations About Program p3.c

```
using namespace std;

#include <iostream>

int main()
{ float b;
  int b = 5;
  return 0;
}
```

- Redeclaration of b with different types
- Not allowed even with the same type

Observations About Program p3.c

```
using namespace std;

#include <iostream>

int main()
{ float b;
  int b = 5;
  return 0;
}
```

- Redeclaration of b with different types
- Not allowed even with the same type
- Semantic error (name and scope analysis)

Observations About Program p4.c

```
using namespace std;

#include <iostream>

int main()
{ int &i;
  cout << i << endl;
  return 0;
}
```

Observations About Program p4.c

```
using namespace std;

#include <iostream>

int main()
{ int &i;
  cout << i << endl;
  return 0;
}
```

- C++ requires references to be initialized

Observations About Program p4.c

```
using namespace std;

#include <iostream>

int main()
{ int &i;
  cout << i << endl;
  return 0;
}
```

- C++ requires references to be initialized
- Cannot be identified by the scanner
- Identified by the parser
  - Token '=' must appear after ID

Observations About Program p4.c

```
using namespace std;

#include <iostream>

int main()
{ int &i;
  cout << i << endl;
  return 0;
}
```

- C++ requires references to be initialized
- Cannot be identified by the scanner
- Identified by the parser
    - Token '=' must appear after ID
- Syntax error and not a semantic error

Observations About Program p5.c

```
using namespace std;

#include <iostream>

int main()
{ short s = 1234567890;
  cout << s << endl;
  return 0;
}
```

Observations About Program p5.c

```
using namespace std;

#include <iostream>

int main()
{ short s = 1234567890;
  cout << s << endl;
  return 0;
}
```

- Overflow

Observations About Program p5.c

```
using namespace std;

#include <iostream>

int main()
{ short s = 1234567890;
  cout << s << endl;
  return 0;
}
```

- Overflow
- Cannot be identified by the scanner
- Cannot be identified by the parser
  - Needs the knowledge of types
  - Needs recording and examining context

Observations About Program p5.c

```
using namespace std;

#include <iostream>

int main()
{ short s = 1234567890;
  cout << s << endl;
  return 0;
}
```

- Overflow
- Cannot be identified by the scanner
- Cannot be identified by the parser
    - Needs the knowledge of types
    - Needs recording and examining context
- Semantic error (type matching)

  Reported as a warning

Observations About Program p6.c

```
using namespace std;

#include <iostream>

int main()
{ int i = 40;
  if ( 1 <= i <= 5)
    cout << " In range\n";
  else
    cout << " Out of Range\n";
  return 0;
}
```

Observations About Program p6.c

```
using namespace std;

#include <iostream>

int main()
{ int i = 40;
  if ( 1 <= i <= 5)
    cout << " In range\n";
  else
    cout << " Out of Range\n";
  return 0;
}
```

- Relational operators are left-associative in C++

  They are non-associative in sclp

Observations About Program p6.c

```
using namespace std;

#include <iostream>

int main()
{ int i = 40;
  if ( 1 <= i <= 5)
    cout << " In range\n";
  else
    cout << " Out of Range\n";
  return 0;
}
```

- Relational operators are left-associative in C++

  They are non-associative in sclp
- `1 <= i` evaluated to `true` whose value is taken as 1 by the compiler

Observations About Program p6.c

```
using namespace std;

#include <iostream>

int main()
{ int i = 40;
  if ( 1 <= i <= 5)
    cout << " In range\n";
  else
    cout << " Out of Range\n";
  return 0;
}
```

- Relational operators are left-associative in C++

  They are non-associative in sclp
- `1 <= i` evaluated to `true` whose value is taken as 1 by the compiler
  - All non-zero integers map to true but true maps only to 1

Observations About Program p6.c

```
using namespace std;

#include <iostream>

int main()
{ int i = 40;
  if ( 1 <= i <= 5)
    cout << " In range\n";
  else
    cout << " Out of Range\n";
  return 0;
}
```

- Relational operators are left-associative in C++

  They are non-associative in sclp
- `1 <= i` evaluated to `true` whose value is taken as 1 by the compiler
  - All non-zero integers map to true but true maps only to 1
- The compiler and run time support cannot know the programmer's intent

  (Does the value of i lie between 1 and 5?)

Observations About Program p6.c

```
using namespace std;

#include <iostream>

int main()
{ int i = 40;
  if ( 1 <= i <= 5)
    cout << " In range\n";
  else
    cout << " Out of Range\n";
  return 0;
}
```

- Relational operators are left-associative in C++

  They are non-associative in sclp
- `1 <= i` evaluated to `true` whose value is taken as 1 by the compiler
  - All non-zero integers map to true but true maps only to 1
- The compiler and run time support cannot know the programmer's intent

  (Does the value of i lie between 1 and 5?)
- Logical error and not a semantic error

Observations About Program p7.c

```
using namespace std;

#include <iostream>

int main()
{ int a[5] = {1, 2, 3, 4,
              5, 6, 7, 8
             };
  return 0;
}
```

Observations About Program p7.c

```
using namespace std;

#include <iostream>

int main()
{ int a[5] = {1, 2, 3, 4,
             5, 6, 7, 8
            };
  return 0;
}
```

- More elements in the initialization than the declared size of the array

Observations About Program p7.c

```
using namespace std;

#include <iostream>

int main()
{ int a[5] = {1, 2, 3, 4,
              5, 6, 7, 8
             };
  return 0;
}
```

- More elements in the initialization than the declared size of the array
- Cannot be identified by the scanner
- Cannot be identified by the parser
    - Requires the knowledge of the size of the array

Observations About Program p7.c

```
using namespace std;

#include <iostream>

int main()
{ int a[5] = {1, 2, 3, 4,
              5, 6, 7, 8
             };
  return 0;
}
```

- More elements in the initialization than the declared size of the array
- Cannot be identified by the scanner
- Cannot be identified by the parser
  - Requires the knowledge of the size of the array
- Semantic error (declaration processing)

Observations About Program p8.c

```
using namespace std;

#include <iostream>

int main()
{ int a[5] = {1, 2, 3};
  int sum;
  for (int i=0; i<10000; i++)
     sum = sum + a[i];
  cout << sum << endl;
  return 0;
}
```

Observations About Program p8.c

```
using namespace std;

#include <iostream>

int main()
{ int a[5] = {1, 2, 3};
  int sum;
  for (int i=0; i<10000; i++)
     sum = sum + a[i];
  cout << sum << endl;
  return 0;
}
```

- Segmentation fault

  Memory access violation

Observations About Program p8.c

```
using namespace std;

#include <iostream>

int main()
{ int a[5] = {1, 2, 3};
  int sum;
  for (int i=0; i<10000; i++)
     sum = sum + a[i];
  cout << sum << endl;
  return 0;
}
```

- Segmentation fault

  Memory access violation

- This is a run time activity and the error cannot be identified by a compiler

Observations About Program p8.c

```
using namespace std;

#include <iostream>

int main()
{ int a[5] = {1, 2, 3};
  int sum;
  for (int i=0; i<10000; i++)
     sum = sum + a[i];
  cout << sum << endl;
  return 0;
}
```

- Segmentation fault
  Memory access violation
- This is a run time activity and the error cannot be identified by a compiler
- Run time error (undefined behaviour)

Observations About Program p8.c

```
using namespace std;

#include <iostream>

int main()
{ int a[5] = {1, 2, 3};
  int sum;
  for (int i=0; i<10000; i++)
     sum = sum + a[i];
  cout << sum << endl;
  return 0;
}
```

- Segmentation fault

  Memory access violation
- This is a run time activity and the error cannot be identified by a compiler
- Run time error (undefined behaviour)
- If we change the loop bound to 5 or 10, memory violation may go undetected, program may not abort, but the result would be unpredictable

  (undefined behaviour)

Observations About Program p8.c

```
using namespace std;

#include <iostream>

int main()
{ int a[5] = {1, 2, 3};
  int sum;
  for (int i=0; i<10000; i++)
     sum = sum + a[i];
  cout << sum << endl;
  return 0;
}
```

- Segmentation fault

  Memory access violation
- This is a run time activity and the error cannot be identified by a compiler
- Run time error (undefined behaviour)
- If we change the loop bound to 5 or 10, memory violation may go undetected, program may not abort, but the result would be unpredictable

  (undefined behaviour)
- If we change the loop bound to 2, it will be a logical error because it is not a memory violation

Observations About Program p9.c

```
using namespace std;
#include <iostream>

int f(int x)
{
  if (x>10) return x;
  else
    if (x>5) return x+5;
}

int main()
{ int i = -5;
  int j = f(i);

  cout << j << endl;
  return 0;
}
```

Observations About Program p9.c

```
using namespace std;
#include <iostream>

int f(int x)
{
  if (x>10) return x;
  else
    if (x>5) return x+5;
}

int main()
{ int i = -5;
  int j = f(i);

  cout << j << endl;
  return 0;
}
```

- Existence of a control flow path along which no value is returned
- Semantic analysis over control flow graph

Observations About Program p9.c

```
using namespace std;
#include <iostream>

int f(int x)
{
  if (x>10) return x;
  else
    if (x>5) return x+5;
}

int main()
{ int i = -5;
  int j = f(i);

  cout << j << endl;
  return 0;
}
```

- Existence of a control flow path along which no value is returned
- Semantic analysis over control flow graph
- A warning to flag a possible undefined behaviour

Observations About Program p9.c

```
using namespace std;
#include <iostream>

int f(int x)
{
  if (x>10) return x;
  else
    if (x>5) return x+5;
}

int main()
{ int i = -5;
  int j = f(i);

  cout << j << endl;
  return 0;
}
```

- Existence of a control flow path along which no value is returned
- Semantic analysis over control flow graph
- A warning to flag a possible undefined behaviour
- What does a language definition say?

  A variable must be *declared* before its use but may not be *defined* before its use

  The latter leads to undefined behaviour

Observations About Program p9.c

control flow graph (CFG) path - Represents possible execution paths in the program. Semantic Analysis is (Before CFG)

```
using namespace std;
#include <iostream>

int f(int x)
{
  if (x>10) return x;
  else
    if (x>5) return x+5;
}

int main()
{ int i = -5;
  int j = f(i);

  cout << j << endl;
  return 0;
}
```

- Existence of a control flow path along which no value is returned
- Semantic analysis over control flow graph
- A warning to flag a possible undefined behaviour
- What does a language definition say?

  A variable must be *declared* before its use but may not be *defined* before its use

  The latter leads to undefined behaviour
- Observe the run time consequences by
  - Add cout statement in f
  - Add x = x + 200 in f
  - Add a call g(y) returning a value in f
  - Change the argument of f to i+2

Observations About Program p10.c

```
using namespace std;
#include <iostream>

int main()
{ float inc = 0.1;
  float sum = 0;
  while (inc != 1.0)
  { sum = sum + inc;
    inc = inc + 0.1;
  }
  cout << sum << endl;
  return 0;
}
```

Observations About Program p10.c

```
using namespace std;
#include <iostream>

int main()
{ float inc = 0.1;
  float sum = 0;
  while (inc != 1.0)
  { sum = sum + inc;
    inc = inc + 0.1;
  }
  cout << sum << endl;
  return 0;
}
```

- Infinite loop?

Observations About Program p10.c

```
using namespace std;
#include <iostream>

int main()
{ float inc = 0.1;
  float sum = 0;
  while (inc != 1.0)
  { sum = sum + inc;
    inc = inc + 0.1;
  }
  cout << sum << endl;
  return 0;
}
```

- Infinite loop?
- Print values in the loop and observe

Observations About Program p10.c

```
using namespace std;
#include <iostream>

int main()
{ float inc = 0.1;
  float sum = 0;
  while (inc != 1.0)
  { sum = sum + inc;
    inc = inc + 0.1;
  }
  cout << sum << endl;
  return 0;
}
```

- Infinite loop?
- Print values in the loop and observe
- Change `!` to `<` and observe

Observations About Program p10.c

```
using namespace std;
#include <iostream>

int main()
{ float inc = 0.1;
  float sum = 0;
  while (inc != 1.0)
  { sum = sum + inc;
    inc = inc + 0.1;
  }
  cout << sum << endl;
  return 0;
}
```

- Infinite loop?
- Print values in the loop and observe
- Change `!` to `<` and observe
- Floating point values are not exact

Observations About Program p10.c

```
using namespace std;
#include <iostream>

int main()
{ float inc = 0.1;
  float sum = 0;
  while (inc != 1.0)
  { sum = sum + inc;
    inc = inc + 0.1;
  }
  cout << sum << endl;
  return 0;
}
```

- Infinite loop?
- Print values in the loop and observe
- Change `!` to < and observe
- Floating point values are not exact
- This is a run time activity and the error cannot be identified by a compiler

Observations About Program p10.c

```
using namespace std;
#include <iostream>

int main()
{ float inc = 0.1;
  float sum = 0;
  while (inc != 1.0)
  { sum = sum + inc;
    inc = inc + 0.1;
  }
  cout << sum << endl;
  return 0;
}
```

- Infinite loop?
- Print values in the loop and observe
- Change `!` to < and observe
- Floating point values are not exact
- This is a run time activity and the error cannot be identified by a compiler
- Logical error and not a semantic error

Observations About Program p11.c

```
using namespace std;
#include <iostream>
short f(short a)
 { cout << " short\n";
   return a;}

long f(long x)
 { cout << " long\n";
   return x;}

char f (char c)
 { cout << " char\n";
   return c;}

int main()
{
 f(100);
}
```

Observations About Program p11.c

```
using namespace std;
#include <iostream>
short f(short a)
 { cout << " short\n";
   return a;}

long f(long x)
 { cout << " long\n";
   return x;}

char f (char c)
 { cout << " char\n";
   return c;}

int main()
{
 f(100);
}
```

- Difficulty in resolving function overloading

Observations About Program p11.c

```
using namespace std;
#include <iostream>
short f(short a)
 { cout << " short\n";
   return a;}

long f(long x)
 { cout << " long\n";
   return x;}

char f (char c)
 { cout << " char\n";
   return c;}

int main()
{
 f(100);
}
```

- Difficulty in resolving function overloading
- Value 100 fits into types char, short, and long

Observations About Program p11.c

```
using namespace std;
#include <iostream>
short f(short a)
 { cout << " short\n";
   return a;}

long f(long x)
 { cout << " long\n";
   return x;}

char f (char c)
 { cout << " char\n";
   return c;}

int main()
{
 f(100);
}
```

- Difficulty in resolving function overloading
- Value 100 fits into types char, short, and long
- Add a function with type int and observe

Observations About Program p11.c

```
using namespace std;
#include <iostream>
short f(short a)
 { cout << " short\n";
   return a;}

long f(long x)
 { cout << " long\n";
   return x;}

char f (char c)
 { cout << " char\n";
   return c;}

int main()
{
 f(100);
}
```

- Difficulty in resolving function overloading
- Value 100 fits into types char, short, and long
- Add a function with type int and observe
- Cannot be identified by the parser
- Semantic error (type matching)

functional overloading error goes away

Observations About Program p12.c

```
using namespace std;
#include <iostream>
long f(long a)
 { cout << " long\n"; return a;}
int f(int x)
 { cout << " int\n"; return x;}
char f (char c)
 { cout << " char\n"; return c;}
int main()
{  short d = 25;
   char ch = '$';
   f(1000000000000);
   f(1234);
   f(ch);
   f(d);
}
```

Observations About Program p12.c

```
using namespace std;
#include <iostream>
long f(long a)
 { cout << " long\n"; return a;}
int f(int x)
 { cout << " int\n"; return x;}
char f (char c)
 { cout << " char\n"; return c;}
int main()
{  short d = 25;
   char ch = '$';
   f(1000000000000);
   f(1234);
   f(ch);
   f(d);
}
```

- Type casting for resolving function overloading

Observations About Program p12.c

```
using namespace std;
#include <iostream>
long f(long a)
 { cout << " long\n"; return a;}
int f(int x)
 { cout << " int\n"; return x;}
char f (char c)
 { cout << " char\n"; return c;}
int main()
{   short d = 25;
    char ch = '$';
    f(1000000000000);
    f(1234);
    f(ch);
    f(d);
}
```

- Type casting for resolving function overloading
- A short value is treated as an int value

Observations About Program p12.c

```
using namespace std;
#include <iostream>
long f(long a)
 { cout << " long\n"; return a;}
int f(int x)
 { cout << " int\n"; return x;}
char f (char c)
 { cout << " char\n"; return c;}
int main()
{   short d = 25;
    char ch = '$';
    f(1000000000000);
    f(1234);
    f(ch);
    f(d);
}
```

this causes issue in windows (long is 32 bits, 64 bits in linux) , hv to define long long function , can't resolve between 3 competing functions with same name

- Type casting for resolving function overloading
- A short value is treated as an int value
- Semantic analysis (type analysis)

Observations About Program p13.c

```
using namespace std;

#include<iostream>

template <class T>
int countzeros (T a[], int size)
{ int count = 0;
  for (int i = 0; i < size; i++)
    if (a[i] == 0) count ++;
  return count;
}
```

```
int main()
{ int x[5]={7, 0 , 5, 1, 0};
  float y[6]={0.0, 1.5, 0.0, 2.5,
              9.5, 0.0005};
  int a=20, b=50, c=-100, d=1000;
  int * p[5]={&a, &b, &c, &d, 0};
  char ch[5]={'a', '0', ',', '0',
              '9'};
  string str[4]={"12", "0", "abc",
                 "0"};
  cout << countzeros(x,5) << endl;
  cout << countzeros(y,6) << endl;
  cout << countzeros(p,5) << endl;
  cout << countzeros(ch,5) << endl;
  cout << countzeros(str,4) << endl;
  return 0;
}
```

Observations About Program p13.c


```
float z = 0.0 ;
cout << (z==0) <<endl;   // cout 1, equality operators work well if floating point has .0* in end
```

- Comparison between string and int not defined
- No zero in array ch   ASCII CODING

int *p[5] is an array of 5 integer pointers storing addresses of a, b, c, d, and nullptr (0).
Values stored: p[0] = &a (20), p[1] = &b (50), p[2] = &c (-100), p[3] = &d (1000), p[4] = nullptr (0).

Observations About Program p14.c

```
using namespace std;

#include<iostream>

template <class T>
int countzeros (T a[], int size)
{ int count = 0;
  for (int i = 0; i < size; i++)
    if (a[i] == 0) count ++;
  return count;
}
```

Observations About Program p14.c

```
using namespace std;

#include<iostream>

template <class T>
int countzeros (T a[], int size)
{ int count = 0;
  for (int i = 0; i < size; i++)
    if (a[i] == 0) count ++;
  return count;
}
```

- No main

Observations About Program p14.c

```
using namespace std;

#include<iostream>

template <class T>
int countzeros (T a[], int size)
{ int count = 0;
  for (int i = 0; i < size; i++)
    if (a[i] == 0) count ++;
  return count;
}
```

- No main
- General error: missing external data function or variable
- Identified by the linker

Observations About Program p14.c

```
using namespace std;

#include<iostream>

template <class T>
int countzeros (T a[], int size)
{ int count = 0;
  for (int i = 0; i < size; i++)
    if (a[i] == 0) count ++;
  return count;
}
```

- No main
- General error: missing external data function or variable
- Identified by the linker
- Using -c option with compilation suppresses the error

Observations About Program p14.c

The compilation phase (-c option) succeeds because the template function is syntactically correct
The linking phase fails because:
There's no main() function, which is required as the program's entry point
The linker can't find any actual usage of the template function
The "missing external data function or variable" error occurs during linking because the program lacks both the required main() function and any concrete instantiation of the template function. This is a classic linking error rather than a compilation error.

```
using namespace std;

#include<iostream>

template <class T>
int countzeros (T a[], int size)
{ int count = 0;
  for (int i = 0; i < size; i++)
    if (a[i] == 0) count ++;
  return count;
}
```

- No main
- General error: missing external data function or variable
- Identified by the linker
- Using -c option with compilation suppresses the error
- Linking error

The key is that transaction.cpp is not meant to be a standalone program - it's a module that will be compiled into an object file (.o) and later linked with other files.
When you compile transaction.cpp:
It only needs to verify that the function definitions match their declarations
It doesn't need main() because it's not a complete program
The -c flag creates just an object file without trying to create

This is different from the previous template example because:
That code was trying to be a complete program (it wasn't marked as a module to be linked later)
The template had no concrete instantiation anywhere
It needed a main() because it was attempting to be a complete program
Think of transaction.cpp like a puzzle piece - it doesn't need to be complete on its own, it just needs to fit correctly with the other pieces when they're all linked together into the final program.

The Role of Semantic Analysis

Examples of Errors

Syntax Directed Definitions

Generating IR

Syntax Directed Translation Schemes

Type Analysis

Name and Scope Analysis

Declaration Processing

Syntax Directed Definitions

Introduction

- Practical compilers use context free grammars to admit a superset of valid sentences and prune out invalid sentences by imposing context sensitive restrictions
- The general strategy is to define and compute some attributes of the symbols of a context free grammar and communicate the semantic information between them through the attributes

Syntax directed attribute evaluation

Syntax Directed Definitions (SDDs)

- A context free grammar augmented with attributes of grammar symbols and semantic rules for evaluating the attributes

$A \rightarrow \alpha$	$b = f(c_1, c_2, \ldots, c_k)$

where b is an attribute of A and $c_i, 1 \leq i \leq k$ are attributes of the symbols in α

- The semantic rules are evaluated when the corresponding grammar rule is used for derivation (in a top down parser) or reduction (in a bottom up parser)
- Notations and conventions
  - For simplicity, we will show attribute evaluation on a parse tree
  - X.attribute refers to the attribute named "attribute" of grammar symbol X
  - Multiple occurrences of a grammar symbol within the same production are distinguished using subscripts

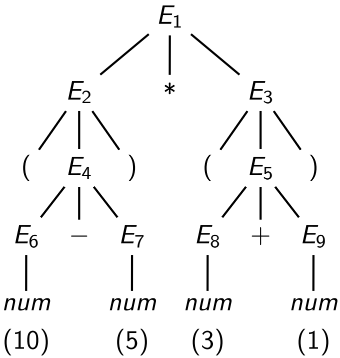
Syntax Directed Definition for Expression Evaluation

- The parser uses the attributes called *value*
- The attribute values for tokens *id* and *num* are supplied by the scanner

$E_1 \rightarrow E_2 \ * \ E_3$	$E_1.value = E_2.value * E_3.value$
$E_1 \rightarrow E_2 \ / \ E_3$	$E_1.value = E_2.value / E_3.value$
$E_1 \rightarrow E_2 \ + \ E_3$	$E_1.value = E_2.value + E_3.value$
$E_1 \rightarrow E_2 \ - \ E_3$	$E_1.value = E_2.value - E_3.value$
$E_1 \rightarrow - \ E_2$	$E_1.value = -E_2.value$
$E_1 \rightarrow (E_2)$	$E_1.value = E_2.value$
$E \rightarrow num$	$E.value = num.value$

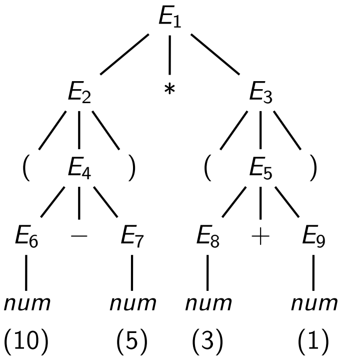
Example of Expression Evaluation

Input expression: $(10 - 5) * (3 + 1)$



Example of Expression Evaluation

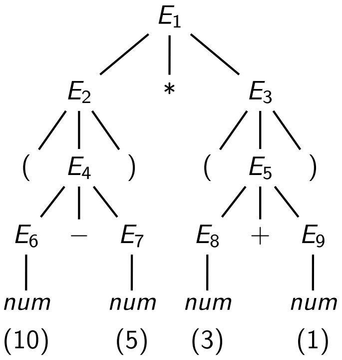
Input expression: $(10 - 5) * (3 + 1)$



$E_6.value$	10
$E_7.value$	5
$E_8.value$	3
$E_9.value$	1

Example of Expression Evaluation

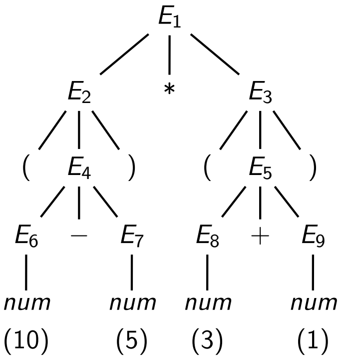
Input expression: $(10-5)*(3+1)$



$E_6.value$	10
$E_7.value$	5
$E_8.value$	3
$E_9.value$	1
$E_4.value$	5

Example of Expression Evaluation

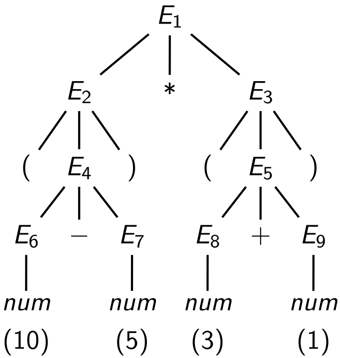
Input expression: $(10 - 5) * (3 + 1)$



$E_6.value$	10
$E_7.value$	5
$E_8.value$	3
$E_9.value$	1
$E_4.value$	5
$E_2.value$	5

Example of Expression Evaluation

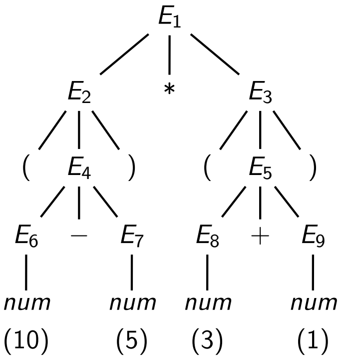
Input expression: $(10 - 5) * (3 + 1)$



$E_6.value$	10
$E_7.value$	5
$E_8.value$	3
$E_9.value$	1
$E_4.value$	5
$E_2.value$	5
$E_5.value$	4

Example of Expression Evaluation

Input expression: $(10 - 5) * (3 + 1)$



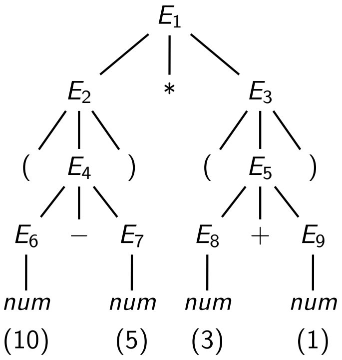
$E_6.value$	10
$E_7.value$	5
$E_8.value$	3
$E_9.value$	1
$E_4.value$	5
$E_2.value$	5
$E_5.value$	4
$E_3.value$	4

The Role of Semantic Analysis
Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

Example of Expression Evaluation

Input expression: $(10 - 5) * (3 + 1)$

left side se start , bottom-up parsing



$E_6.value$	10
$E_7.value$	5
$E_8.value$	3
$E_9.value$	1
$E_4.value$	5
$E_2.value$	5
$E_5.value$	4
$E_3.value$	4
$E_1.value$	20

SDDs for Generating IR

SDDs for Generating IR

- Generating IR for unary and binary expressions
- Generating IR for ternary expression
- Generating IR for WHILE loop
- Generating IR for array accesses
- Generating IR for field accesses in structures
- Generating IR for field accesses through pointers

SDD for Generating IR for Expression

- Input statement. $x = (a - b) * (c + d)$
- Desired output

$$\boxed{\begin{array}{l} t_0 = a - b \\ t_1 = c + d \\ t_2 = t_0 * t_1 \\ x = t_2 \end{array}}$$

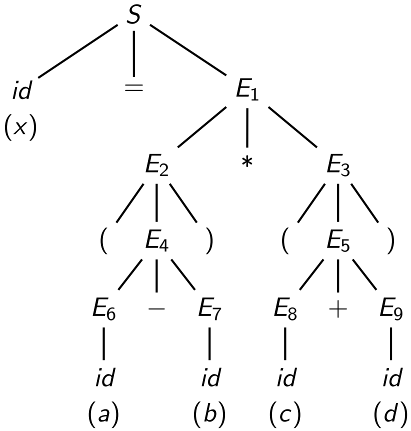
SDD for Generating IR for Expression

- We use attributes called *name* (value supplied by the scanner), *place* (the source or the temporary variable that holds the result), and *code*
- Function *gen* generates code for an assignment statement, function *expr* generates the code for an expression, function *getNewTemp* returns the name of a new temporary, and operator $||$ concatenates code

$S \rightarrow id = E$	$c_1 = gen(id.place, =, E.place)$ $S.code = E.code \mid\mid c_1$
$E_1 \rightarrow E_2\ op\ E_3$	$t_1 = getNewTemp(\,);$ $c_1 = E_2.code;\ \ c_2 = E_3.code$ $c_3 = gen(t_1, =, expr(E_2.place, op, E_3.place))$ $E_1.code = c_1 \mid\mid c_2 \mid\mid c_3$ $E_1.place = t_1$
$E_1 \rightarrow (E_2)$	$E_1.code = E_2.code$ $E_1.place = E_2.place$
$E \rightarrow id$	$E.code = NULL$ $E.place = id.name$

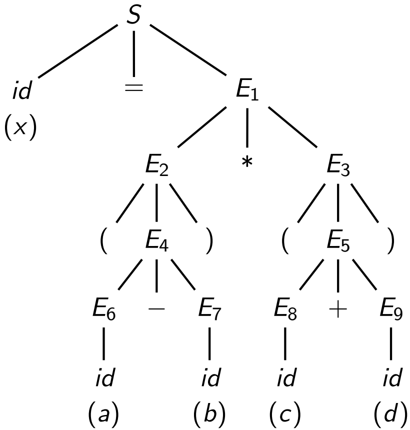
Example of Generating IR for Expression

Input statement: $x = (a - b) * (c + d)$



Example of Generating IR for Expression

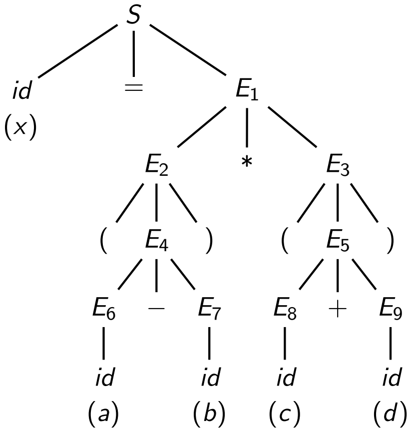
Input statement: $x = (a - b) * (c + d)$



$E_6.place$	a
$E_7.place$	b
$E_8.place$	c
$E_9.place$	d

Example of Generating IR for Expression

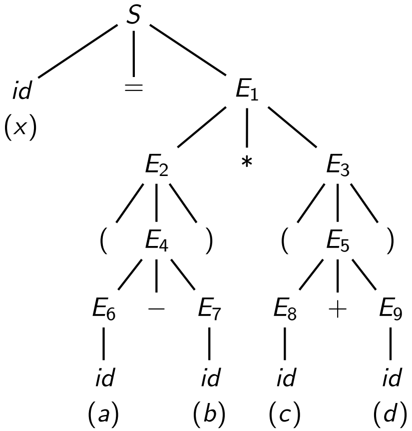
Input statement: $x = (a - b) * (c + d)$



$E_6.place$	a
$E_7.place$	b
$E_8.place$	c
$E_9.place$	d
$E_4.place$, $E_2.place$ $E_4.code$, $E_2.code$	t_0 $t_0 = a - b$

Example of Generating IR for Expression

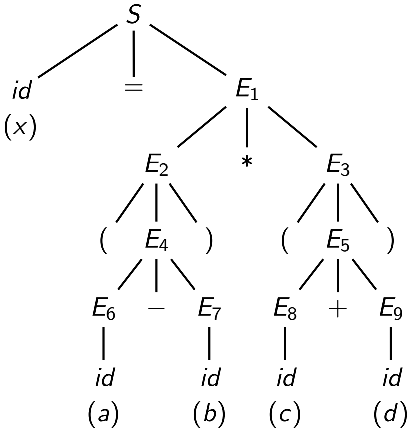
Input statement: $x = (a - b) * (c + d)$



$E_6.place$	a
$E_7.place$	b
$E_8.place$	c
$E_9.place$	d
$E_4.place, E_2.place$ $E_4.code, E_2.code$	t_0 $t_0 = a - b$
$E_5.place, E_3.place$ $E_5.code, E_3.code$	t_1 $t_1 = c + d$

Example of Generating IR for Expression

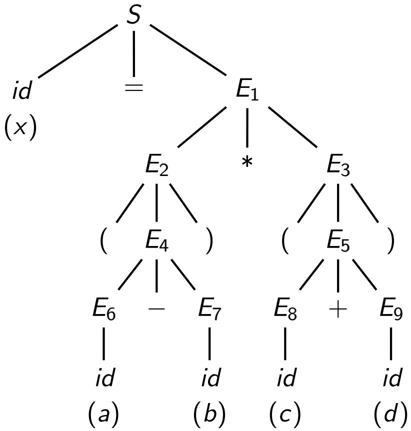
Input statement: $x = (a - b) * (c + d)$



$E_6.place$	a
$E_7.place$	b
$E_8.place$	c
$E_9.place$	d
$E_4.place$, $E_2.place$ $E_4.code$, $E_2.code$	t_0 $t_0 = a - b$
$E_5.place$, $E_3.place$ $E_5.code$, $E_3.code$	t_1 $t_1 = c + d$
$E_1.place$	t_2
$E_1.code$	$t_0 = a - b$ $t_1 = c + d$ $t_2 = t_0 * t_1$

Example of Generating IR for Expression

Input statement: $x = (a - b) * (c + d)$



$E_6.place$	a
$E_7.place$	b
$E_8.place$	c
$E_9.place$	d
$E_4.place, E_2.place$ $E_4.code, E_2.code$	t_0 $t_0 = a - b$
$E_5.place, E_3.place$ $E_5.code, E_3.code$	t_1 $t_1 = c + d$
$E_1.place$	t_2
$E_1.code$	$t_0 = a - b$ $t_1 = c + d$ $t_2 = t_0 * t_1$
$S.code$	$t_0 = a - b$ $t_1 = c + d$ $t_2 = t_0 * t_1$ $x = t_2$

Designing SDD for Generating IR for Ternary Expression

$$E_1 \rightarrow E_2 \; ? \; E_3 \; : \; E_4$$

$E_1.place = t_2$
$E_1.code$

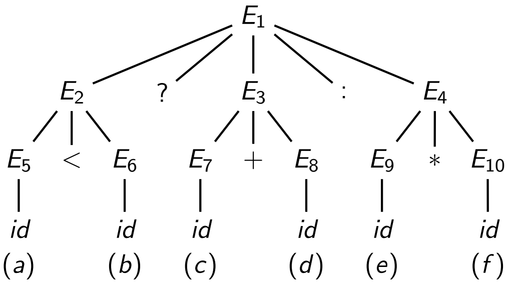
	$\boxed{E_2.code}$
	$t_1 = \neg E_2.place$
	if t_1 goto l_1
	$\boxed{E_3.code}$
	$t_2 = E_3.place$
	goto l_2
l_1 :	$\boxed{E_4.code}$
	$t_2 = E_4.place$
l_2 :	

SDD for Generating IR for Ternary Expression

For simplicity, we view the IR as strings and arguments of *gen* as strings without showing the construction of strings explicitly

$E_1 \rightarrow E_2 \;?\; E_3 \;:\; E_4$	$t_1 = getNewTemp(\,);\; t_2 = getNewTemp(\,)$ $l_1 = getNewLabel(\,);\; l_2 = getNewLabel(\,)$ $c_1 = E_2.code \,\|\|\, gen(t_1 = \neg E_2.place) \,\|\|\, gen(\text{if } t_1 \text{ goto } l_1)$ $c_2 = E_3.code \,\|\|\, gen(t_2 = E_3.place) \,\|\|\, gen(\text{goto } l_2)$ $c_3 = gen(l_1:) \,\|\|\, E_4.code \,\|\|\, gen(t_2 = E_4.place)$ $c_4 = gen(l_2:)$ $E_1.code = c_1 \,\|\|\, c_2 \,\|\|\, c_3 \,\|\|\, c_4$ $E_1.place = t_2$

Example of Generating IR for Ternary Expression



$E_5.place$	a
$E_6.place$	b
$E_7.place$	c
$E_8.place$	d
$E_9.place$	e
$E_{10}.place$	f

$E_2.place$ $E_2.code$	t_0 $t_0 = a < b$
$E_3.place$ $E_3.code$	t_1 $t_1 = c + d$
$E_4.place$ $E_4.code$	t_2 $t_2 = e * f$

$E_1.place$		t_4
$E_1.code$	c_1	$t_0 = a < b$ $t_3 = !t_0$ if t_3 goto l_1
	c_2	$t_1 = c + d$ $t_4 = t1$ goto l_2
	c_3	$l_1 :$ $t_2 = e * f$ $t_4 = t2$
	c_4	$l_2 :$

SDD for Generating IR for WHILE loop

$$S_1 \rightarrow \text{WHILE} \ (\ E \ ) \ S_2$$

$S_1.code$

$l_1:$ $\boxed{E.code}$
$t_1 = \neg E_2.place$
if t_1 goto l_2
$\boxed{S_2.code}$
goto l_1
$l_2:$

$S_1 \rightarrow \text{WHILE} \ (\ E \ ) \ S_2$	$t_1 = getNewTemp(\,);$ $l_1 = getNewLabel(\,);\ l_2 = getNewLabel(\,)$ $c_1 = gen(l_1:) \mid\mid E.code$ $c_2 = gen(t_1 = \neg E.place) \mid\mid gen(\text{if } t_1 \text{ goto } l_2)$ $c_3 = S_2.code \mid\mid gen(\text{goto } l_1)$ $c_4 = gen(l_2:)$ $S_1.code = c_1 \mid\mid c_2 \mid\mid c_3 \mid\mid c_4$

Undefined Behaviour of Pre/Post Increment/Decrement in C

- For expression E_1 *op* E_2,
  - E_1 and E_2 may be evaluated in any order (unspecified behaviour)
  - E_1 and E_2 must be evaluated before evaluating *op*
- For `++i + ++i`, the order of evaluation of the two occurrences is unspecified

  This leads to unpredictable results implying undefined behaviour

GCC's Handling of Pre/Post Increment/Decrement in C

```
#include <stdio.h>

int main()
{
   int i,j;
   i = -1;
   j = [              ];
   printf ("%d\n",j);
   return 0;
}
```

GCC's Handling of Pre/Post Increment/Decrement in C

```
#include <stdio.h>

int main()
{
   int i,j;
   i = -1;
   j = [              ];
   printf ("%d\n",j);
   return 0;
}
```

Expression	Result
`i + (i + (++i + ++i))`	4

GCC's Handling of Pre/Post Increment/Decrement in C

```
#include <stdio.h>

int main()
{
    int i,j;
    i = -1;
    j = [          ];
    printf ("%d\n",j);
    return 0;
}
```

Expression	Result
`i + (i + (++i + ++i))`	4
`i + (i + 1 + (++i + ++i))`	3

GCC's Handling of Pre/Post Increment/Decrement in C

```
#include <stdio.h>

int main()
{
    int i,j;
    i = -1;
    j = [          ];
    printf ("%d\n",j);
    return 0;
}
```

first operands are evaluated fully, left assosciatively before doing operation

Expression	Result
`i + (i + (++i + ++i))`	4
`i + ((i + 1) + (++i + ++i))`	3
`(i + 1) + ((i + 1) + (++i + ++i))`	2

The value decreases with addition of 1!

Modelling GCC's Handling of Pre/Post Increment/Decrement

$E \rightarrow ++\ id$	
$E \rightarrow --\ id$	
$E \rightarrow id\ ++$	
$E \rightarrow id\ --$	

Modelling GCC's Handling of Pre/Post Increment/Decrement

$E \rightarrow ++\ id$	$c_1 = gen(id.name, =, expr(id.name, +, 1))$ $E_1.code = c_1$ $E_1.place = id.name$
$E \rightarrow --\ id$	
$E \rightarrow id\ ++$	
$E \rightarrow id\ --$	

Modelling GCC's Handling of Pre/Post Increment/Decrement

IIT Bombay
cs302: Implementation of Programming Languages

Topic:
Semantic Analysis

Section:

The Role of Semantic Analysis

Examples of Errors

Syntax Directed Definitions

Generating IR

Syntax Directed Translation Schemes

Type Analysis

Name and Scope Analysis

Declaration Processing

$E \rightarrow ++\ id$	$c_1 = gen(id.name, =, expr(id.name, +, 1))$ $E_1.code = c_1$ $E_1.place = id.name$
$E \rightarrow --\ id$	$c_1 = gen(id.name, =, expr(id.name, -, 1))$ $E_1.code = c_1$ $E_1.place = id.name$
$E \rightarrow id\ ++$	
$E \rightarrow id\ --$	

Modelling GCC's Handling of Pre/Post Increment/Decrement

$E \rightarrow ++\ id$	$c_1 = gen(id.name, =, expr(id.name, +, 1))$ $E_1.code = c_1$ $E_1.place = id.name$
$E \rightarrow --\ id$	$c_1 = gen(id.name, =, expr(id.name, -, 1))$ $E_1.code = c_1$ $E_1.place = id.name$
$E \rightarrow id\ ++$	$t_1 = getNewTemp(\,);$ $c_1 = gen(t_1, =, id.name)$ $c_2 = gen(id.name, =, expr(id.name, +, 1))$ $E_1.code = c_1 \parallel c_2$ $E.place = t_1$
$E \rightarrow id\ --$	$t_1 = getNewTemp(\,);$ $c_1 = gen(t_1, =, id.name)$ $c_2 = gen(id.name, =, expr(id.name, -, 1))$ $E_1.code = c_1 \parallel c_2$ $E.place = t_1$

Modelling GCC's Handling of Pre/Post Increment/Decrement

$E \rightarrow ++\ id$	$c_1 = gen(id.name, =, expr(id.name, +, 1))$ $E_1.code = c_1$ $E_1.place = id.name$
$E \rightarrow --\ id$	$c_1 = gen(id.name, =, expr(id.name, -, 1))$ $E_1.code = c_1$ $E_1.place = id.name$
	$t_1 = getNewTemp(\,);$ $c_1 = gen(t_1, =, id.name)$ $c_2 = gen(id.name, =, expr(id.name, +, 1))$ $E_1.code = c_1 \mid\mid c_2$ $E.place = t_1$
$E \rightarrow id\ --$	$t_1 = getNewTemp(\,);$ $c_1 = gen(t_1, =, id.name)$ $c_2 = gen(id.name, =, expr(id.name, -, 1))$ $E_1.code = c_1 \mid\mid c_2$ $E.place = t_1$

A statement to increment/decrement the *id* is generated

Modelling GCC's Handling of Pre/Post Increment/Decrement

$E \rightarrow ++\ id$	$c_1 = gen(id.name, =, expr(id.name, +, 1))$ $E_1.code = c_1$ $E_1.place = id.name$
$E \rightarrow --\ id$	$c_1 = gen(id.name, =, expr($… $E_1.code = c_1$ $E_1.place = id.name$
$E \rightarrow id\ ++$	$t_1 = getNewTemp(\,);$ $c_1 = gen(t_1, =, id.name)$ $c_2 = gen(id.name, =, expr(id.name, +, 1))$ $E_1.code = c_1 \mid\mid c_2$ $E.place = t_1$
$E \rightarrow id\ --$	$t_1 = getNewTemp(\,);$ $c_1 = gen(t_1, =, id.nan$… $c_2 = gen(id.name, =,$… $E_1.code = c_1 \mid\mid c_2$ $E.place = t_1$

For pre increment/decrement, $E.place$ is the name of the id

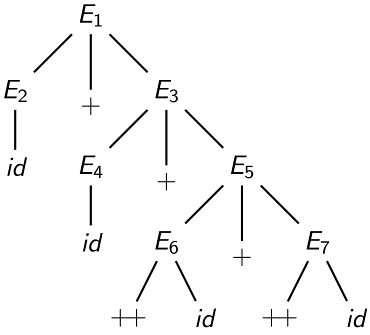
For post increment/decrement, $E.place$ is a temporary storing the value before increment/decrement

Modelling GCC's Handling of Pre/Post Increment/Decrement

$E \rightarrow ++\ id$	$c_1 = gen(id.name, =, expr(id.name, +, 1))$ $E_1.code = c_1$ $E_1.place = id.name$
$E \rightarrow --\ id$	$c_1 = gen(id.name, =, expr(id.name, -, 1))$ $E_1.code = c_1$ $E_1.place = id.name$
$E \rightarrow id\ ++$	$t_1 = getNewTemp(\,);$ $c_1 = gen(t_1, =, id.name)$ $c_2 = gen(id.name, =, expr(id.name, +, 1))$ $E_1.code = c_1 \mid\mid c_2$ $E.place = t_1$
$E \rightarrow id\ --$	$t_1 = getNewTemp(\,);$ $c_1 = gen(t_1, =, id.name)$ $c_2 = gen(id.name, =, expr(id.name, -, 1))$ $E_1.code = c_1 \mid\mid c_2$ $E.place = t_1$

Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```

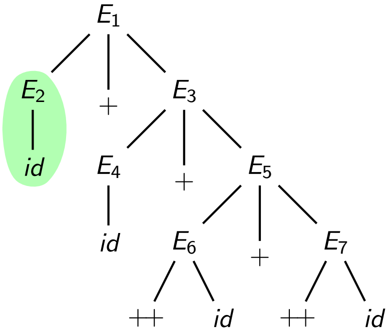


Modelling GCC's Handling of Pre/Post Increment/Decrement


```
i + (i + (++i + ++i))
```

$E_2.code$ $NULL$

$E_2.place$ i

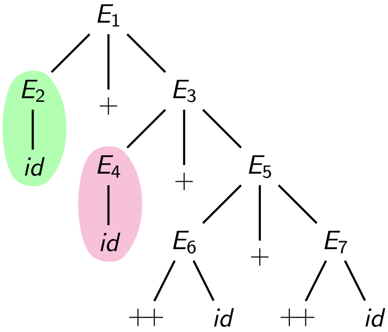


Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```

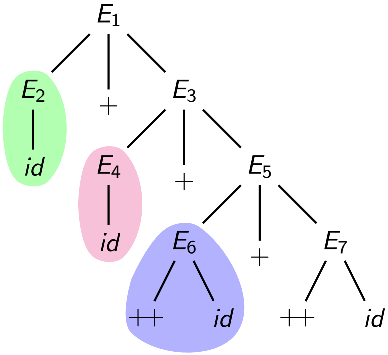
$E_2.code$ $NULL$
$E_2.place$ i

$E_4.code$ $NULL$
$E_4.place$ i



Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```



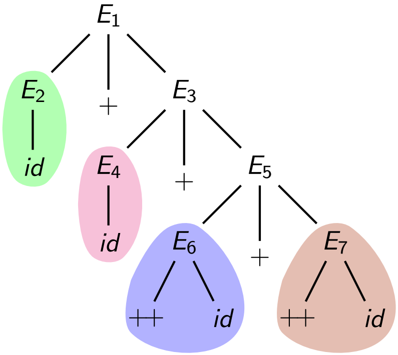
$E_2.code$ $NULL$
$E_2.place$ i

$E_4.code$ $NULL$
$E_4.place$ i

$E_6.code$ $i = i + 1$
$E_6.place$ i

Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```



$E_2.code$ $NULL$
$E_2.place$ i

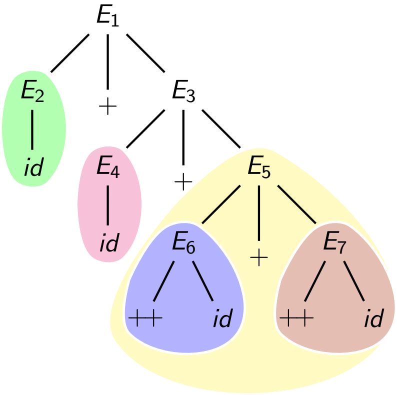
$E_4.code$ $NULL$
$E_4.place$ i

$E_6.code$ $i = i + 1$
$E_6.place$ i

$E_7.code$ $i = i + 1$
$E_7.place$ i

Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```



$E_2.code$ $NULL$
$E_2.place$ i

$E_4.code$ $NULL$
$E_4.place$ i

$E_6.code$ $i = i + 1$
$E_6.place$ i

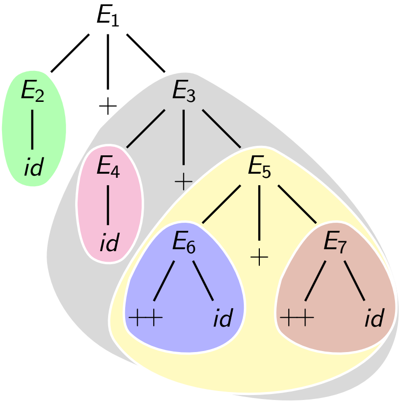
$E_7.code$ $i = i + 1$
$E_7.place$ i

$E_5.code$
$i = i + 1$
$i = i + 1$
$t_0 = i + i$

$E_5.place$ t_0

Modelling GCC's Handling of Pre/Post Increment/Decrement


```
i + (i + (++i + ++i))
```



$E_2.code$ NULL
$E_2.place$ i

$E_4.code$ NULL
$E_4.place$ i

$E_6.code$ $i = i + 1$
$E_6.place$ i

$E_7.code$ $i = i + 1$
$E_7.place$ i

$E_5.code$
$i = i + 1$
$i = i + 1$
$t_0 = i + i$

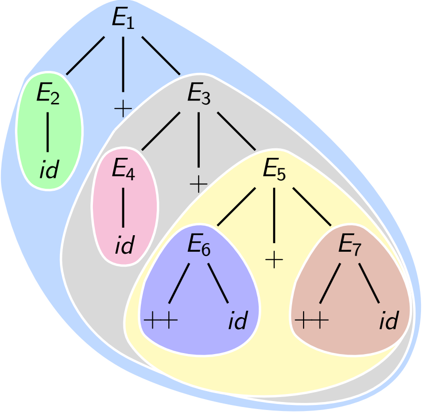
$E_5.place$ t_0

$E_3.code$
$i = i + 1$
$i = i + 1$
$t_0 = i + i$
$t_1 = i + t_0$

$E_3.place$ t_1

Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```



$E_2.code$ NULL
$E_2.place$ i

$E_4.code$ NULL
$E_4.place$ i

$E_6.code$ $i = i + 1$
$E_6.place$ i

$E_7.code$ $i = i + 1$
$E_7.place$ i

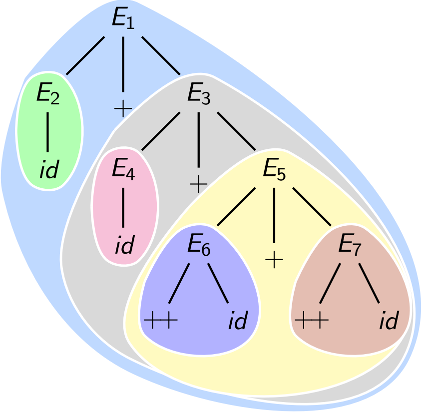
$E_5.code$
$$\begin{array}{l} i = i + 1 \\ i = i + 1 \\ \hline t_0 = i + i \end{array}$$
$E_5.place$ t_0

$E_3.code$
$$\begin{array}{l} i = i + 1 \\ i = i + 1 \\ t_0 = i + i \\ \hline t_1 = i + t_0 \end{array}$$
$E_3.place$ t_1

$E_1.code$
$$\begin{array}{l} i = i + 1 \\ i = i + 1 \\ t_0 = i + i \\ t_1 = i + t_0 \\ \hline t_2 = i + t_1 \end{array}$$
$E_1.place$ t_2

Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```



Values of variables during execution

i	-1
t_0	
t_1	
t_2	

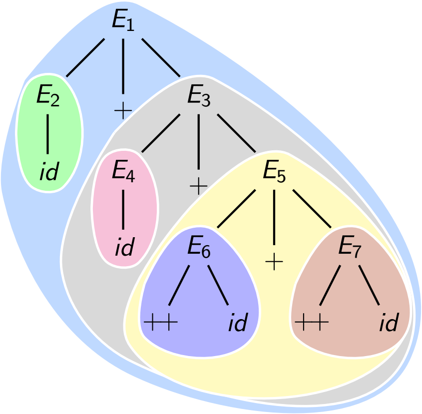
$E_1.code$

$$
\begin{array}{l}
i = i + 1 \\
i = i + 1 \\
t_0 = i + i \\
t_1 = i + t_0 \\
\hline
t_2 = i + t_1
\end{array}
$$

$E_1.place\ t_2$

Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```



Values of variables during execution

i	0
t_0	
t_1	
t_2	

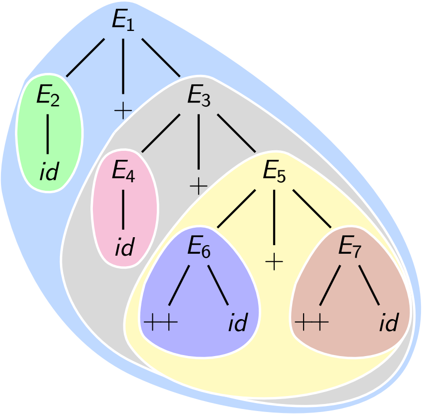
$E_1.code$

$i = i + 1$
$i = i + 1$
$t_0 = i + i$
$t_1 = i + t_0$
$t_2 = i + t_1$

$E_1.place\ t_2$

Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```



Values of variables during execution

i	1
t_0	
t_1	
t_2	

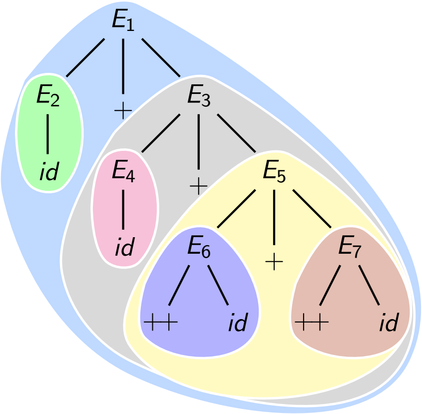
$E_1.code$

$i = i + 1$
$i = i + 1$
$t_0 = i + i$
$t_1 = i + t_0$
$t_2 = i + t_1$

$E_1.place$ t_2

Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```



Values of variables during execution

i	1
t_0	2
t_1	
t_2	

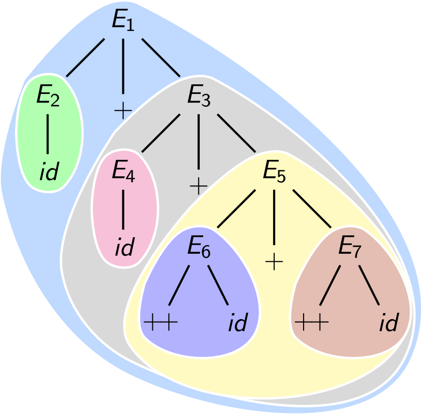
$E_1.code$

$i = i + 1$
$i = i + 1$
$t_0 = i + i$
$t_1 = i + t_0$
$t_2 = i + t_1$

$E_1.place$ t_2

Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```



Values of variables during execution

i	1
t_0	2
t_1	3
t_2	

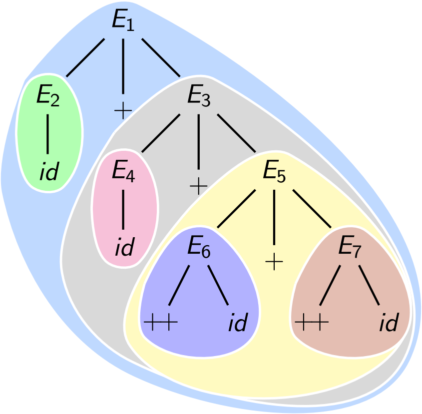
$E_1.code$

$i = i + 1$
$i = i + 1$
$t_0 = i + i$
$t_1 = i + t_0$
$t_2 = i + t_1$

$E_1.place$ t_2

Modelling GCC's Handling of Pre/Post Increment/Decrement

```
i + (i + (++i + ++i))
```



Values of variables during execution

i	1
t_0	2
t_1	3
t_2	4

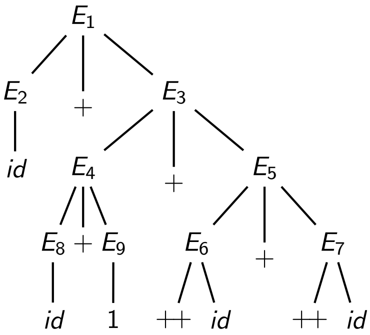
$E_1.code$

$i = i + 1$
$i = i + 1$
$t_0 = i + i$
$t_1 = i + t_0$
$t_2 = i + t_1$

$E_1.place$ t_2

Code for Updated Expression

```
i + (i + 1 + (++i + ++i))
```



For ease of comparison, we retain the labels of the parse tree nodes by adding new expressions nodes E_8 and E_9 even if they appear out of sequence in parsing

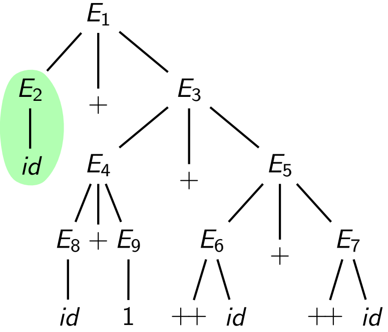
We also retain the numbering of temporaries and use t_3 for the new temporary although it is the first temporary to be generated

Code for Updated Expression

```
i + (i + 1 + (++i + ++i))
```

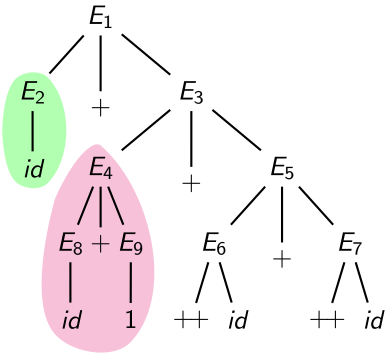
$E_2.code$ $NULL$

$E_2.place$ i



Code for Updated Expression


```
i + (i + 1 + (++i + ++i))
```



$E_2.code$ $NULL$

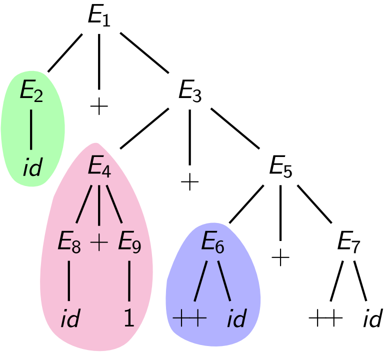
$E_2.place$ i

$E_4.code$ $t_3 = i + 1$

$E_4.place$ t_3

Code for Updated Expression

```
i + (i + 1 + (++i + ++i))
```



$E_2.code$ $NULL$

$E_2.place$ i

$E_4.code$ $t_3 = i + 1$

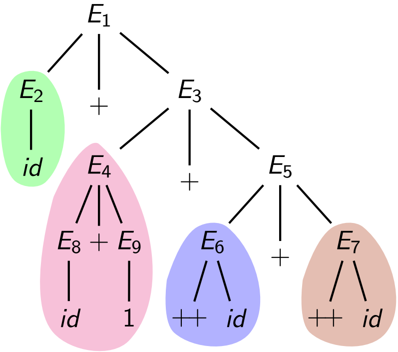
$E_4.place$ t_3

$E_6.code$ $i = i + 1$

$E_6.place$ i

Code for Updated Expression

```
i + (i + 1 + (++i + ++i))
```



$E_2.code$ $NULL$
$E_2.place$ i

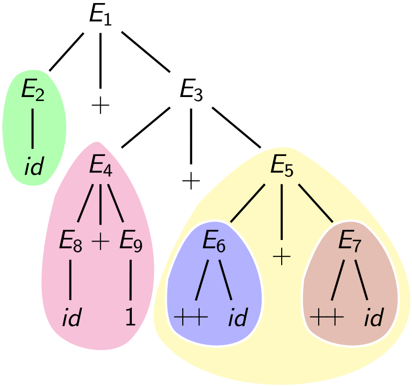
$E_4.code$ $t_3 = i + 1$
$E_4.place$ t_3

$E_6.code$ $i = i + 1$
$E_6.place$ i

$E_7.code$ $i = i + 1$
$E_7.place$ i

Code for Updated Expression

```
i + (i + 1 + (++i + ++i))
```



$E_2.code$ $NULL$
$E_2.place$ i

$E_4.code$ $t_3 = i + 1$
$E_4.place$ t_3

$E_6.code$ $i = i + 1$
$E_6.place$ i

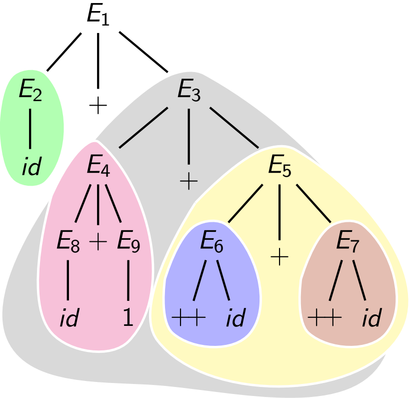
$E_7.code$ $i = i + 1$
$E_7.place$ i

$E_5.code$ $\begin{array}{l} i = i + 1 \\ i = i + 1 \\ \hline t_0 = i + i \end{array}$

$E_5.place$ t_0

Code for Updated Expression

```
i + (i + 1 + (++i + ++i))
```



$E_2.code$ NULL
$E_2.place$ i

$E_4.code$ $t_3 = i + 1$
$E_4.place$ t_3

$E_6.code$ $i = i + 1$
$E_6.place$ i

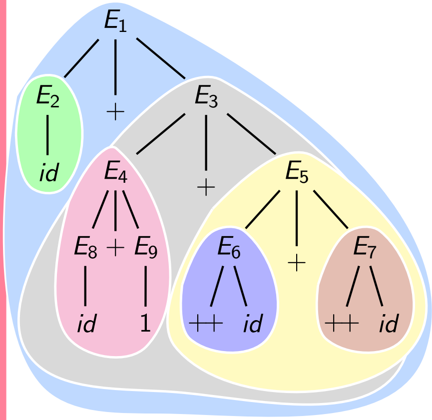
$E_7.code$ $i = i + 1$
$E_7.place$ i

$E_5.code$
$$\begin{array}{l} i = i + 1 \\ i = i + 1 \\ \hline t_0 = i + i \end{array}$$
$E_5.place$ t_0

$E_3.code$
$$\begin{array}{l} t_3 = i + 1 \\ i = i + 1 \\ i = i + 1 \\ t_0 = i + i \\ \hline t_1 = t_3 + t_0 \end{array}$$
$E_3.place$ t_1

Code for Updated Expression

```
i + (i + 1 + (++i + ++i))
```



$E_2.code$ NULL
$E_2.place$ i

$E_4.code$ $t_3 = i + 1$
$E_4.place$ t_3

$E_6.code$ $i = i + 1$
$E_6.place$ i

$E_7.code$ $i = i + 1$
$E_7.place$ i

$E_5.code$
$$\begin{array}{l} i = i + 1 \\ i = i + 1 \\ \hline t_0 = i + i \end{array}$$
$E_5.place$ t_0

$E_3.code$
$$\begin{array}{l} t_3 = i + 1 \\ i = i + 1 \\ i = i + 1 \\ t_0 = i + i \\ \hline t_1 = t_3 + t_0 \end{array}$$
$E_3.place$ t_1

$E_1.code$
$$\begin{array}{l} t_3 = i + 1 \\ i = i + 1 \\ i = i + 1 \\ t_0 = i + i \\ t_1 = t_3 + t_0 \\ \hline t_2 = i + t_1 \end{array}$$
$E_1.place$ t_2

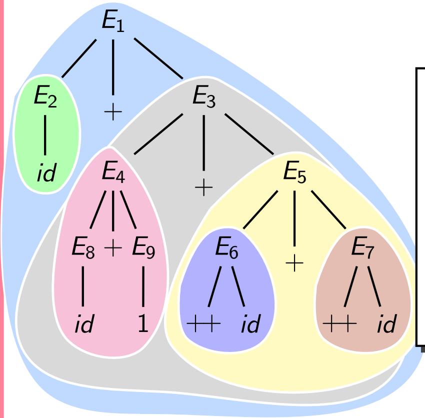
IIT Bombay
cs302: Implementation of Programming Languages

Topic:
Semantic Analysis

Section:
The Role of Semantic Analysis
Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

Code for Updated Expression

```
i + (i + 1 + (++i + ++i))
```



$E_2.code$ $NULL$
$E_2.place$ i

$E_4.code$ $t_3 = i + 1$
$E_4.place$ t_3

$E_6.code$ $i = i + 1$

$E_3.code$
$$\begin{array}{l} t_3 = i + 1 \\ i = i + 1 \\ i = i + 1 \\ t_0 = i + i \\ \hline t_1 = t_3 + t_0 \end{array}$$
$E_3.place$ t_1

Now $E_4.code$ is not $NULL$ and computes t_3 *before* i is incremented for E_6 and E_7

Evaluation E_3 uses t_3 and not the (twice incremented) i as its left operand

de
$$\begin{array}{l} t_3 = i + 1 \\ i = i + 1 \\ i = i + 1 \\ t_0 = i + i \\ t_1 = t_3 + t_0 \\ \hline t_2 = i + t_1 \end{array}$$
$E_1.place$ t_2

$E_5.place$ t_0

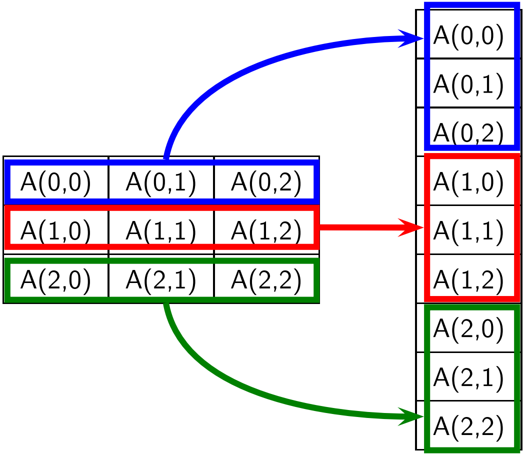
Representing Arrays in Memory

A 2-D Array	Row Major Representation	Column Major Representation

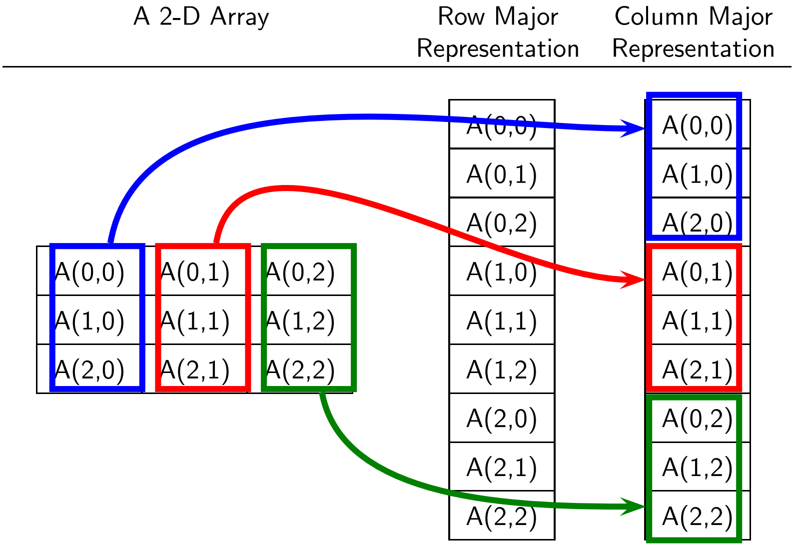
A(0,0)	A(0,1)	A(0,2)
A(1,0)	A(1,1)	A(1,2)
A(2,0)	A(2,1)	A(2,2)

Representing Arrays in Memory

A 2-D Array	Row Major Representation	Column Major Representation

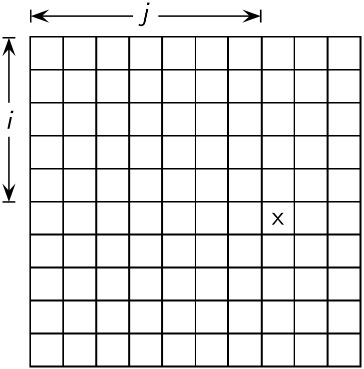


Representing Arrays in Memory



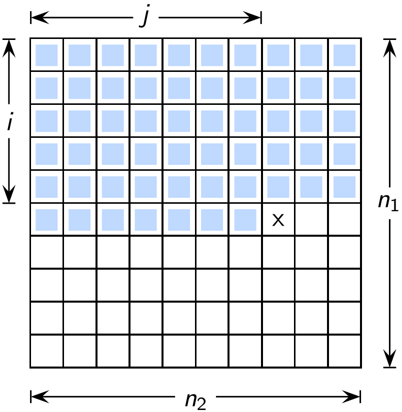
Array Address Calculation

Cell (i, j)



Array Address Calculation

Cell (i, j)



- Indices begin at 0 (0, 1, 2,. . .)
- Array is stored in the row major form
- The starting address of the cell is

$$\text{Base} + (i \times n_2) + j$$

- The number of cells in the first dimension does not matter

Array Address Calculation

- Consider a 2-D array with limits (n_1, n_2)
  - The offset (i.e., starting address) of an element (i_1, i_2) is

$$\boxed{i_1 \times n_2 + i_2}$$

Array Address Calculation

- Consider a 2-D array with limits (n_1, n_2)
    - The offset (i.e., starting address) of an element (i_1, i_2) is

$$\boxed{i_1 \times n_2 + i_2}$$

- Consider a k-D array with limits $(n_1, n_2, \ldots, n_k)$
    - The offset (i.e., starting address) of an element $(i_1, i_2, \ldots, i_k)$ is

$$\boxed{((((i_1 \times n_2 + i_2) \times n_3 + i_3) \times n_4 + i_4) \ldots) \times n_k + i_k}$$

Note that n_1 does not appear in the expression

Array Address Calculation

- Consider a 2-D array with limits (n_1, n_2)
  - The offset (i.e., starting address) of an element (i_1, i_2) is

$$\boxed{i_1 \times n_2 + i_2}$$

- Consider a k-D array with limits $(n_1, n_2, \ldots, n_k)$
  - The offset (i.e., starting address) of an element $(i_1, i_2, \ldots, i_k)$ is

$$\boxed{((((i_1 \times n_2 + i_2) \times n_3 + i_3) \times n_4 + i_4) \ldots) \times n_k + i_k}$$

  Note that n_1 does not appear in the expression
  - It can be obtained from the recurrence

$$\boxed{\begin{aligned} O_1 &= i_1 \\ O_{j+1} &= O_j \times n_{j+1} + i_{j+1} \end{aligned}}$$

  where O_m gives the expression for dimension $1 \leq m \leq k$

Example of Array Address Calculation

Address calculation formula

$$\boxed{\begin{aligned} O_1 &= i_1 \\ O_{j+1} &= O_j \times n_{j+1} + i_{j+1} \end{aligned}}$$

Declaration

```
int b[10][20][30];
```

Access

```
a = b[c][d*e][f+g];
```

Generated code

$$\begin{aligned}
t_1 &= c * 20 && // \quad O_1 \times n_2 \\
t_2 &= d * e && // \quad i_2 \\
t_3 &= t_1 + t_2 && // \quad O_2 = O_1 \times n_2 + i_2 \\
t_4 &= t_3 * 30 && // \quad O_2 \times n_3 \\
t_5 &= f + g && // \quad i_3 \\
t_6 &= t_4 + t_5 && // \quad O_3 = O_2 \times n_3 + i_3 \\
t_7 &= t_6 * 4 && // \quad O_3 \times \texttt{sizeof(int)} \\
t_8 &= b[t_7] \\
a &= t_8
\end{aligned}$$

SDD for Generating Code for Array Accesses

We use the following attributes

- $S.code$, $E.place$, $E.code$, $id.name$, and $num.value$
- $A.name$: name of the array
- $A.offset$: name of the variable holding the offset of A
- $A.code$: code that access array element
- $A.ndim$: dimension number being considered

We use the following functions apart from $gen(\cdot)$ and $getNewTemp(\,)$ functions

- $width(A)$ gives the number of bytes required an element in the array
- $dimLimit(A, i)$ gives the number of elements in dimension i (i.e., n_i)

SDD for Generating Code for Array Accesses

$S \rightarrow id = E$ $E \rightarrow id$ $E \rightarrow num$ $E \rightarrow \ldots$	

SDD for Generating Code for Array Accesses

$S \rightarrow id = E$ $E \rightarrow id$ $E \rightarrow num$ $E \rightarrow \ldots$	
$E \rightarrow A$	

SDD for Generating Code for Array Accesses

$S \rightarrow id = E$ $E \rightarrow id$ $E \rightarrow num$ $E \rightarrow \ldots$	
$E \rightarrow A$	
$A \rightarrow id\,[\,E\,]$	

SDD for Generating Code for Array Accesses

$S \rightarrow id = E$ $E \rightarrow id$ $E \rightarrow num$ $E \rightarrow \ldots$	
$E \rightarrow A$	
$A \rightarrow id\,[\,E\,]$	
$A_1 \rightarrow A_2\,[\,E\,]$	

SDD for Generating Code for Array Accesses

$S \rightarrow id = E$ $E \rightarrow id$ $E \rightarrow num$ $E \rightarrow \ldots$	// The usual rules
$E \rightarrow A$	
$A \rightarrow id\,[\,E\,]$	
$A_1 \rightarrow A_2\,[\,E\,]$	

SDD for Generating Code for Array Accesses

$S \rightarrow id = E$ $E \rightarrow id$ $E \rightarrow num$ $E \rightarrow \ldots$	// The usual rules
$E \rightarrow A$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $c_1 = gen(t_1, =, A.offset \times width(A.name))$ $c_2 = gen(t_2, =, A.name, [, t_1,])$ $E.code = A.code \mid\mid c_1 \mid\mid c_2$ $E.place = t_2$
$A \rightarrow id\,[\,E\,]$	
$A_1 \rightarrow A_2\,[\,E\,]$	

SDD for Generating Code for Array Accesses

$S \rightarrow id = E$ $E \rightarrow id$ $E \rightarrow num$ $E \rightarrow \ldots$	// The usual rules
$E \rightarrow A$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $c_1 = gen(t_1, =, A.offset \times width(A.name))$ $c_2 = gen(t_2, =, A.name, [, t_1,])$ $E.code = A.code \mid\mid c_1 \mid\mid c_2$ $E.place = t_2$
$A \rightarrow id\,[\,E\,]$	$A.name = id.name;\ A.ndim = 1$ $A.offset = E.place;\ A.code = E.code$
$A_1 \rightarrow A_2\,[\,E\,]$	

SDD for Generating Code for Array Accesses

$S \rightarrow id = E$ $E \rightarrow id$ $E \rightarrow num$ $E \rightarrow \ldots$	// The usual rules
$E \rightarrow A$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $c_1 = gen(t_1, =, A.offset \times width(A.name))$ $c_2 = gen(t_2, =, A.name, [, t_1,])$ $E.code = A.code \mid\mid c_1 \mid\mid c_2$ $E.place = t_2$
$A \rightarrow id\,[\,E\,]$	$A.name = id.name;\ A.ndim = 1$ $A.offset = E.place;\ A.code = E.code$
$A_1 \rightarrow A_2\,[\,E\,]$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $A_1.name = A_2.name;\ A_1.ndim = A_2.ndim + 1$ $c_1 = gen(t_1, =, A_2.offset \times dimLimit(A_1.name, A_1.ndim))$ $c_2 = gen(t_2, =, t_1, +, E.place)$ $A_1.code = A_2.code \mid\mid E.code \mid\mid c_1 \mid\mid c_2$ $A_1.offset = t_2$

For example, translating arr[i+1] might produce:
t1 = 4 * (i+1) (assuming 4-byte elements)
t2 = arr[t1]
Final code combines these operations, with t2 holding the result

SDD for Generating Code for Array Accesses

$S \rightarrow id = E$ $E \rightarrow id$ $E \rightarrow num$ $E \rightarrow \ldots$	// The usual rules $\quad \boxed{O_j \times n_{j+1}} + \boxed{i_{j+1}}$
$E \rightarrow A$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $c_1 = gen(t_1, =, A.offset \times width(A.name))$ $c_2 = gen(t_2, =, A.name, [, t_1,])$ $E.code = A.code \mid\mid c_1 \mid\mid c_2$ $E.place = t_2$
$A \rightarrow id\,[\,E\,]$	$A.name = id.name;\ A.ndim = 1$ $A.offset = E.place;\ A.code = E.code$
$A_1 \rightarrow A_2\,[\,E\,]$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $A_1.name = A_2.name;\ A_1.ndim = A_2.ndim + 1$ $c_1 = gen(t_1, =, A_2.offset \times dimLimit(A_1.name, A_1.ndim))$ $c_2 = gen(t_2, =, t_1, +, E.place)$ $A_1.code = A_2.code \mid\mid E.code \mid\mid c_1 \mid\mid c_2$ $A_1.offset = t_2$

SDD for Generating Code for Array Accesses

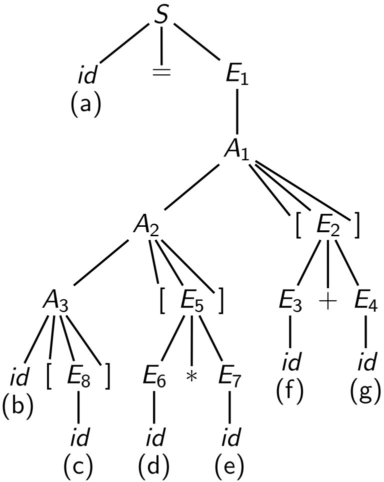
$S \rightarrow id = E$ $E \rightarrow id$ $E \rightarrow num$ $E \rightarrow \ldots$	// The usual rules
$E \rightarrow A$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $c_1 = gen(t_1, =, A.offset \times width(A.name))$ $c_2 = gen(t_2, =, A.name, [, t_1,])$ $E.code = A.code \mid\mid c_1 \mid\mid c_2$ $E.place = t_2$
$A \rightarrow id\,[\,E\,]$	$A.name = id.name;\ A.ndim = 1$ $A.offset = E.place;\ A.code = E.code$
$A_1 \rightarrow A_2\,[\,E\,]$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $A_1.name = A_2.name;\ A_1.ndim = A_2.ndim + 1$ $c_1 = gen(t_1, =, A_2.offset \times dimLimit(A_1.name, A_1.ndim))$ $c_2 = gen(t_2, =, t_1, +, E.place)$ $A_1.code = A_2.code \mid\mid E.code \mid\mid c_1 \mid\mid c_2$ $A_1.offset = t_2$

Example of Generating Code for Array Accesses

Declaration	`int b[10][20][30];`
Access	`a = b[c][d*e][f+g];`

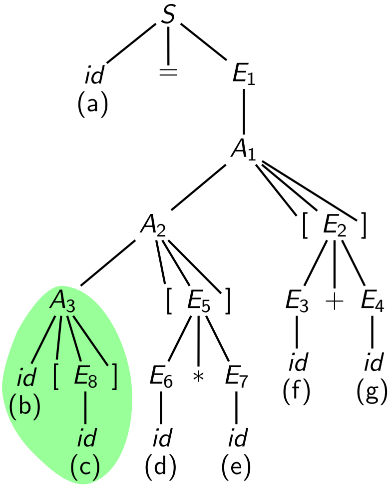
Example of Generating Code for Array Accesses

Declaration	`int b[10][20][30];`
Access	`a = b[c][d*e][f+g];`



Example of Generating Code for Array Accesses

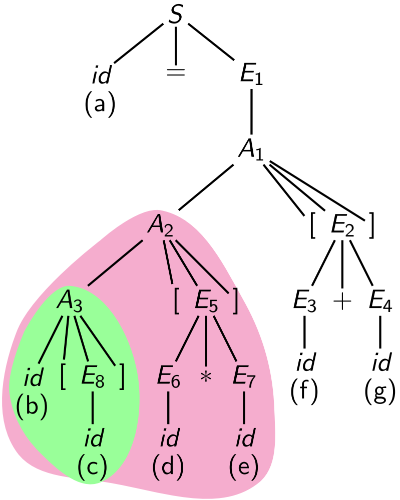
Declaration	`int b[10][20][30];`
Access	`a = b[c][d*e][f+g];`



$E_8.code$	$NULL$
$E_8.place$	c
$A_3.name$	b
$A_3.ndim$	1
$A_3.code$	$NULL$
$A_3.offset$	c

Example of Generating Code for Array Accesses

Declaration	`int b[10][20][30];`
Access	`a = b[c][d*e][f+g];`

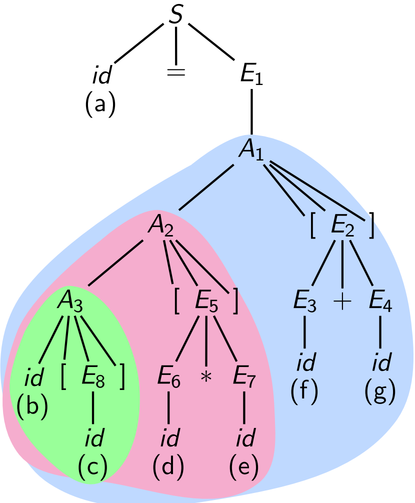


$E_8.code$	$NULL$
$E_8.place$	c
$A_3.name$	b
$A_3.ndim$	1
$A_3.code$	$NULL$
$A_3.offset$	c

$E_5.code$	$t_1 = d * e$
$E_5.place$	t_1
$A_2.name$	b
$A_2.ndim$	2
$A_2.code$	$t_1 = d * e$ $t_2 = c * 20$ $t_3 = t_2 + t_1$
$A_2.offset$	$t3$

Example of Generating Code for Array Accesses

Declaration	`int b[10][20][30];`
Access	`a = b[c][d*e][f+g];`



$E_5.code$ $\quad t_1 = d * e$

$E_5.place$ $\quad t_1$

$A_2.name$ $\quad b$

$A_2.ndim$ $\quad 2$

$A_3.code$ $\quad t_1 = d * e$; $t_2 = c * 20$; $t_3 = t_2 + t_1$

$A_3.offset$ $\quad t3$

$E_2.code$ $\quad t_4 = f + g$

$E_2.place$ $\quad t_4$

$A_1.name$ $\quad b$

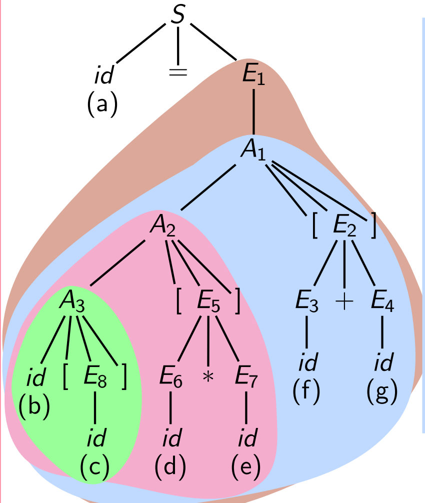
$A_1.ndim$ $\quad 3$

$A_1.code$ $\quad t_1 = d * e$; $t_2 = c * 20$; $t_3 = t_2 + t_1$; $t_4 = f + g$; $t_5 = t_3 * 30$; $t_6 = t_5 + t_4$

$A_1.offset$ $\quad t6$

Example of Generating Code for Array Accesses

Declaration	`int b[10][20][30];`
Access	`a = b[c][d*e][f+g];`



$E_2.code$ $t_4 = f + g$

$E_2.place$ t_4

$A_1.name$ b

$A_1.ndim$ 3

$A_1.code$

$t_1 = d * e$
$t_2 = c * 20$
$t_3 = t_2 + t_1$
$t_4 = f + g$
$t_5 = t_3 * 30$
$t_6 = t_5 + t_4$

$A_1.offset$ $t6$

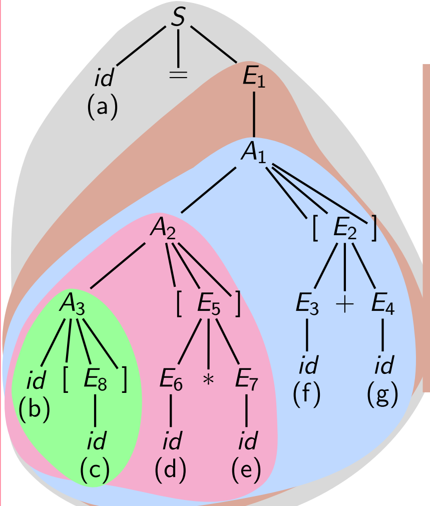
$E_1.code$

$t_1 = d * e$
$t_2 = c * 20$
$t_3 = t_2 + t_1$
$t_4 = f + g$
$t_5 = t_3 * 30$
$t_6 = t_5 + t_4$
$t_7 = t_6 * 4$
$t_8 = b[t_7]$

$E_1.place$ t_8

Example of Generating Code for Array Accesses

Declaration	`int b[10][20][30];`
Access	`a = b[c][d*e][f+g];`



$E_1.code$

$$
\begin{aligned}
t_1 &= d * e\\
t_2 &= c * 20\\
t_3 &= t_2 + t_1\\
t_4 &= f + g\\
t_5 &= t_3 * 30\\
t_6 &= t_5 + t_4\\
\hline
t_7 &= t_6 * 4\\
t_8 &= b[t_7]
\end{aligned}
$$

$E_1.place$ t_8

$S.code$

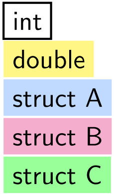
$$
\begin{aligned}
t_1 &= d * e\\
t_2 &= c * 20\\
t_3 &= t_2 + t_1\\
t_4 &= f + g\\
t_5 &= t_3 * 30\\
t_6 &= t_5 + t_4\\
t_7 &= t_6 * 4\\
t_8 &= b[t_7]\\
\hline
a &= t_8
\end{aligned}
$$

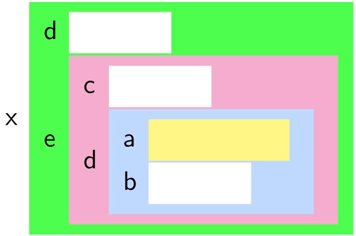
Generating IR for Field Accesses in a Structure: Approach 1

```
struct A { double a; int b; };

struct B { int c; struct A d; };

struct C { int d; struct B e; };

struct C x;

int b = x.e.d.b;
```

Generating IR for Field Accesses in a Structure: Approach 1

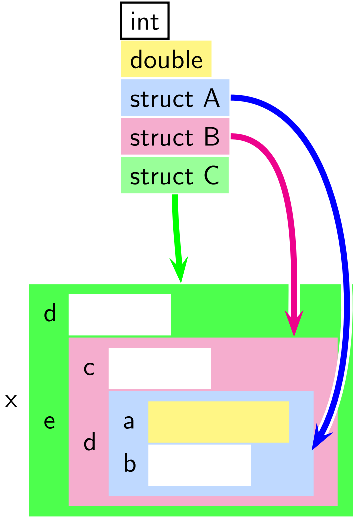
```
struct A { double a; int b; };

struct B { int c; struct A d; };

struct C { int d; struct B e; };

struct C x;

int b = x.e.d.b;
```





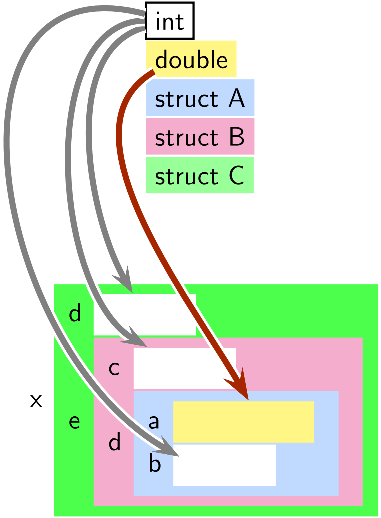
Generating IR for Field Accesses in a Structure: Approach 1

```
struct A { double a; int b; };

struct B { int c; struct A d; };

struct C { int d; struct B e; };

struct C x;

int b = x.e.d.b;
```



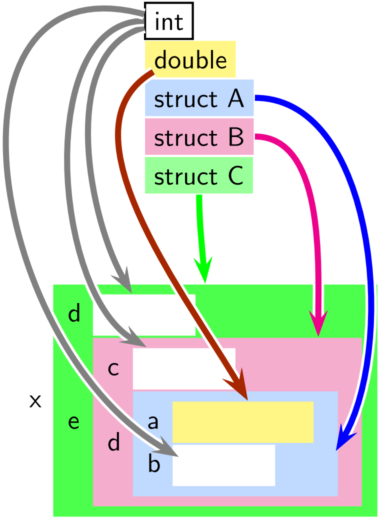
Generating IR for Field Accesses in a Structure: Approach 1

```
struct A { double a; int b; };

struct B { int c; struct A d; };

struct C { int d; struct B e; };

struct C x;

int b = x.e.d.b;
```



Generating IR for Field Accesses in a Structure: Approach 1

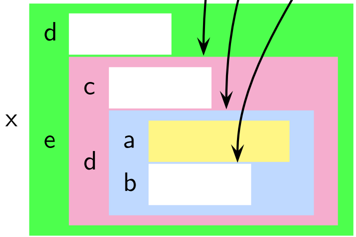
```
struct A { double a; int b; };

struct B { int c; struct A d; };

struct C { int d; struct B e; };

struct C x;

int b = x.e.d.b;
```



Generating IR for Field Accesses in a Structure: Approach 1

```
struct A { double a; int b; };

struct B { int c; struct A d; };

struct C { int d; struct B e; };

struct C x;

int b = x.e.d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8



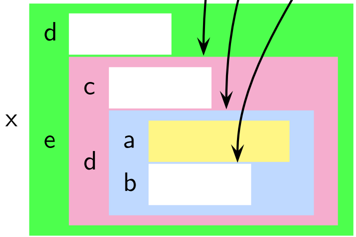
Generating IR for Field Accesses in a Structure: Approach 1

```
struct A { double a; int b; };

struct B { int c; struct A d; };

struct C { int d; struct B e; };

struct C x;

int b = x.e.d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8

Required IR code

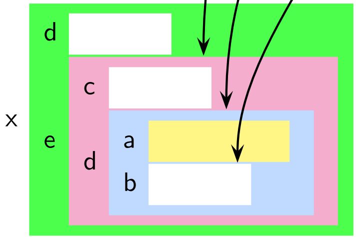
$$t_1 = \&x$$
$$t_2 = t_1 + 16$$
$$t_3 = *t_2$$
$$b = t_3$$



Generating IR for Field Accesses in a Structure: Approach 1

```
struct A { double a; int b; };

struct B { int c; struct A d; };

struct C { int d; struct B e; };

struct C x;

int b = x.e.d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8



Required IR code

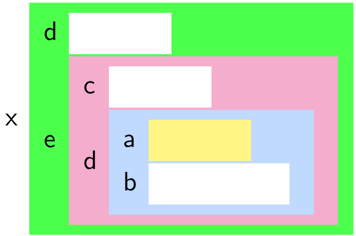
$$t_1 = \&x$$
$$t_2 = t_1 + 16$$
$$t_3 = *t_2$$
$$b = t_3$$

Generating IR for Field Accesses in a Structure: Approach 2

```
struct A { double a; int b; };

struct B { int c; struct A d; };

struct C { int d; struct B e; };

struct C x;

int b = x.e.d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8

$$
\begin{aligned}
t_1 &= \&x \\
t_2 &= t_1 + 4 \quad //\&x.e \\
t_3 &= t_2 + 4 \quad //\&x.e.d \\
t_4 &= t_3 + 8 \quad //\&x.e.d.b \\
t_5 &= *t_4 \\
b &= t_5
\end{aligned}
$$



SDD for Generating Code for Field Accesses: Approach 1

We use the following attributes

- $S.code$, $E.place$, $E.code$, $id.name$, and $num.value$
- $F.struct$: name of the structure variable
- $F.offset$: offset of the field accessed using F
  (used to reach the address of the field)
- $F.type$: type of the field accessed using F
  $pointer(\tau)$ denotes the type of a pointer to type τ

This approach computes the final offsets at compile time and hence uses $F.offset$ attribute but not $F.code$ attribute

We use the following functions apart from $gen(\cdot)$ and $getNewTemp(\,)$

- $offset(\tau, f)$ gives the offset of field f in structure type τ
- $type(\tau, f)$ gives the type of field f in structure type τ

SDD for Generating Code for Field Accesses: Approach 1

$S \rightarrow id = E$	
$E \rightarrow id$	
$E \rightarrow \ldots$	

SDD for Generating Code for Field Accesses: Approach 1

$S \rightarrow id = E$	
$E \rightarrow id$	
$E \rightarrow \ldots$	
$E \rightarrow F$	

SDD for Generating Code for Field Accesses: Approach 1

$S \rightarrow id = E$	
$E \rightarrow id$	
$E \rightarrow \ldots$	
$E \rightarrow F$	
$F \rightarrow id_1 \cdot id_2$	

SDD for Generating Code for Field Accesses: Approach 1

$S \to id = E$	
$E \to id$	
$E \to \ldots$	
$E \to F$	
$F \to id_1 \cdot id_2$	
$F_1 \to F_2 \cdot id$	

SDD for Generating Code for Field Accesses: Approach 1

$S \rightarrow id = E$	$S.code = E.code \ \| \ gen(id.place, =, E.place)$
$E \rightarrow id$	$E.code = NULL; E.place = id.name$
$E \rightarrow \ldots$	// The usual rules
$E \rightarrow F$	
$F \rightarrow id_1 \cdot id_2$	
$F_1 \rightarrow F_2 \cdot id$	

SDD for Generating Code for Field Accesses: Approach 1

$S \rightarrow id = E$	$S.code = E.code \;\|\|\; gen(id.place, =, E.place)$
$E \rightarrow id$	$E.code = NULL$; $E.place = id.name$
$E \rightarrow \ldots$	// The usual rules
$E \rightarrow F$	$t_1 = getNewTemp(\,)$; $t_2 = getNewTemp(\,)$; $t_3 = getNewTemp(\,)$ $c_1 = gen(t_1, =, \&F.struct)$ $c_2 = gen(t_2, =, t_1 + F.offset)$ $c_3 = gen(t_3, =, *t_2)$ $E.code = c_1 \;\|\|\; c_2 \;\|\|\; c_3$ $E.place = t_3$
$F \rightarrow id_1 \cdot id_2$	
$F_1 \rightarrow F_2 \cdot id$	

IIT Bombay
cs302: Implementation of Programming Languages

Topic:
Semantic Analysis

Section:
The Role of Semantic Analysis
Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

SDD for Generating Code for Field Accesses: Approach 1

$S \rightarrow id = E$	$S.code = E.code \;\|\|\; gen(id.place, =, E.place)$
$E \rightarrow id$	$E.code = NULL; E.place = id.name$
$E \rightarrow \ldots$	// The usual rules
$E \rightarrow F$	$t_1 = getNewTemp(\,); t_2 = getNewTemp(\,); t_3 = getNewTemp(\,)$ $c_1 = gen(t_1, =, \&F.struct)$ $c_2 = gen(t_2, =, t_1 + F.offset)$ $c_3 = gen(t_3, =, *t_2)$ $E.code = c_1 \;\|\|\; c_2 \;\|\|\; c_3$ $E.place = t_3$
$F \rightarrow id_1 \cdot id_2$	$F.struct = id_1.name$ $F.type = pointer(type(id_1.type, id_2.name))$ $F.offset = offset(id_1.type, id_2.name)$
$F_1 \rightarrow F_2 \cdot id$	

SDD for Generating Code for Field Accesses: Approach 1

$S \rightarrow id = E$	$S.code = E.code \;\|\|\; gen(id.place, =, E.place)$
$E \rightarrow id$	$E.code = NULL; E.place = id.name$
$E \rightarrow \ldots$	// The usual rules
$E \rightarrow F$	$t_1 = getNewTemp(\,); t_2 = getNewTemp(\,); t_3 = getNewTemp(\,)$ $c_1 = gen(t_1, =, \&F.struct)$ $c_2 = gen(t_2, =, t_1 + F.offset)$ $c_3 = gen(t_3, =, *t_2)$ $E.code = c_1 \;\|\|\; c_2 \;\|\|\; c_3$ $E.place = t_3$
$F \rightarrow id_1 \cdot id_2$	$F.struct = id_1.name$ $F.type = pointer(type(id_1.type, id_2.name))$ $F.offset = offset(id_1.type, id_2.name)$
$F_1 \rightarrow F_2 \cdot id$	$F_1.struct = F_2.struct$ $F_1.type = pointer(type(F_2.type, id.name))$ $F_1.offset = F_2.offset + offset(F_2.type, id.name)$

Example of Generating Code for Field Accesses: Approach 1

Field Access

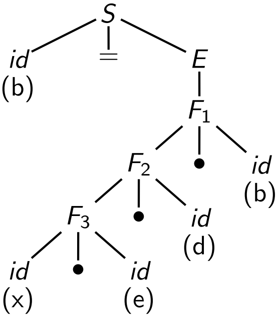
```
b = x.e.d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8

Example of Generating Code for Field Accesses: Approach 1

Field Access

```
b = x.e.d.b;
```



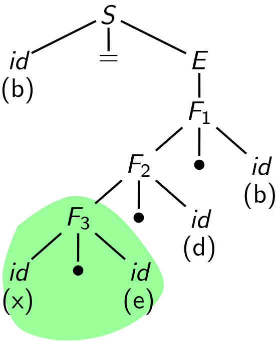
Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8

Example of Generating Code for Field Accesses: Approach 1

Field Access

```
b = x.e.d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8

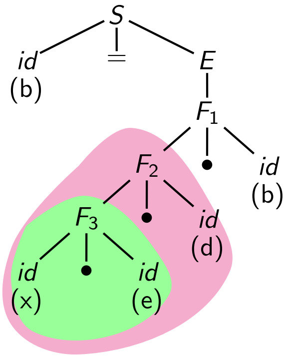


$F_3.struct$ x
$F_3.type$ $struct\ B*$
$F_3.offset$ 4

Example of Generating Code for Field Accesses: Approach 1

Field Access

```
b = x.e.d.b;
```



Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8

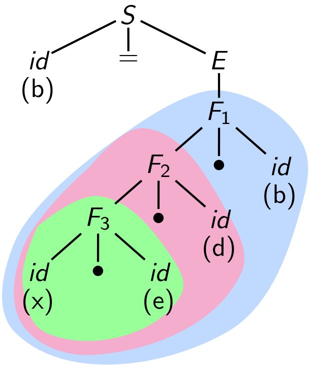
$F_3.struct$ x
$F_3.type$ $struct\ B*$
$F_3.offset$ 4

$F_2.struct$ x
$F_2.type$ $struct\ A*$
$F_2.offset$ 8

Example of Generating Code for Field Accesses: Approach 1

Field Access

```
b = x.e.d.b;
```



Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8

$F_3.struct$ x
$F_3.type$ $struct\ B*$
$F_3.offset$ 4

$F_2.struct$ x
$F_2.type$ $struct\ A*$
$F_2.offset$ 8

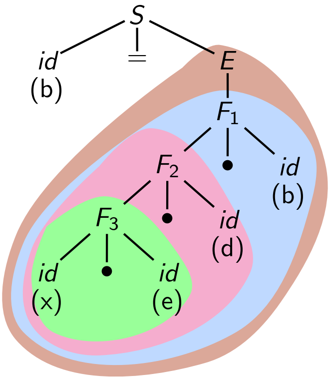
$F_1.struct$ x
$F_1.type$ $int*$
$F_1.offset$ 16

Example of Generating Code for Field Accesses: Approach 1

Field Access

```
b = x.e.d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8



$F_3.struct$ x
$F_3.type$ $struct\ B*$
$F_3.offset$ 4

$F_2.struct$ x
$F_2.type$ $struct\ A*$
$F_2.offset$ 8

$F_1.struct$ x
$F_1.type$ $int*$
$F_1.offset$ 16

$E.code$
$$t_1 = \&x$$
$$t_2 = t_1 + 16$$
$$t_3 = *t_2$$

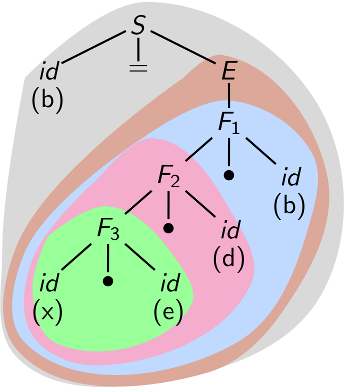
$E.place$ t_3

Example of Generating Code for Field Accesses: Approach 1

Field Access

```
b = x.e.d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8



$F_3.struct$ x
$F_3.type$ $struct\ B*$
$F_3.offset$ 4

$F_2.struct$ x
$F_2.type$ $struct\ A*$
$F_2.offset$ 8

$F_1.struct$ x
$F_1.type$ $int*$
$F_1.offset$ 16

$E.code$
$$t_1 = \&x$$
$$t_2 = t_1 + 16$$
$$t_3 = *t_2$$
$E.place$ t_3

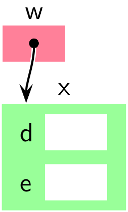
$S.code$
$$t_1 = \&x$$
$$t_2 = t_1 + 16$$
$$t_3 = *t_2$$
$$b = t_3$$

IR for Field Accesses Through Pointers: Example 1

```
struct A { double a; int b; };
struct B { int c; struct A *d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = &z;
int b = w->e->d->b;
```

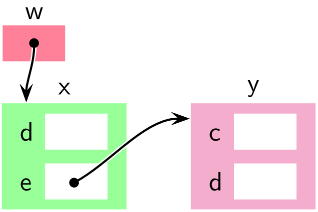
IR for Field Accesses Through Pointers: Example 1

```
struct A { double a; int b; };
struct B { int c; struct A *d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = &z;
int b = w->e->d->b;
```



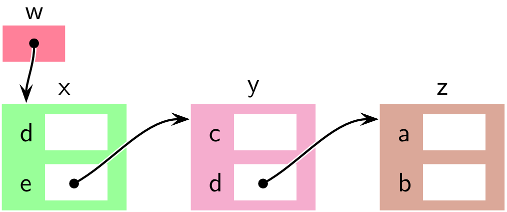
IR for Field Accesses Through Pointers: Example 1

```
struct A { double a; int b; };
struct B { int c; struct A *d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = &z;
int b = w->e->d->b;
```



IR for Field Accesses Through Pointers: Example 1

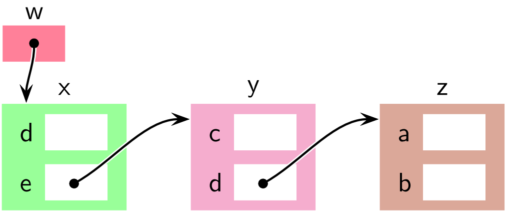
```
struct A { double a; int b; };
struct B { int c; struct A *d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = &z;
int b = w->e->d->b;
```



IR for Field Accesses Through Pointers: Example 1

```
struct A { double a; int b; };
struct B { int c; struct A *d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = &z;
int b = w->e->d->b;
```

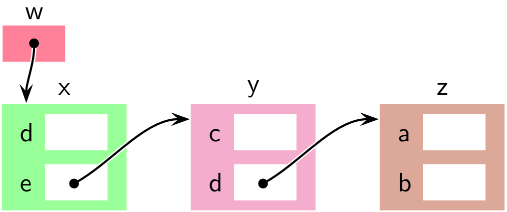
Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8



IR for Field Accesses Through Pointers: Example 1

```
struct A { double a; int b; };
struct B { int c; struct A *d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = &z;
int b = w->e->d->b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8



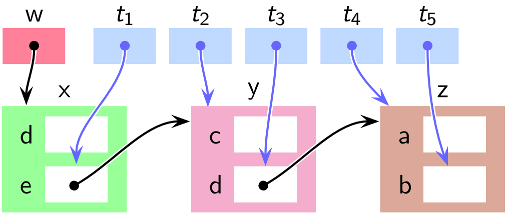
IR code for access expression `w->e->d->b`

$$\begin{aligned}
t_1 &= w + 4 & //\&(x.e)\\
t_2 &= *t_1 & //\&y\\
t_3 &= t_2 + 4 & //\&(y.d)\\
t_4 &= *t_3 & //\&z\\
t_5 &= t_4 + 8 & //\&(z.b)\\
t_6 &= *t_5 &
\end{aligned}$$

IR for Field Accesses Through Pointers: Example 1

```
struct A { double a; int b; };
struct B { int c; struct A *d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = &z;
int b = w->e->d->b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8



IR code for access expression `w->e->d->b`

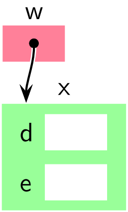
$$t_1 = w + 4 \quad //\&(x.e)$$
$$t_2 = *t_1 \quad //\&y$$
$$t_3 = t_2 + 4 \quad //\&(y.d)$$
$$t_4 = *t_3 \quad //\&z$$
$$t_5 = t_4 + 8 \quad //\&(z.b)$$
$$t_6 = *t_5$$

IR for Field Accesses Through Pointers: Example 2

```
struct A { double a; int b; };
struct B { int c; struct A d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = z;
int b = w->e->d.b;
```

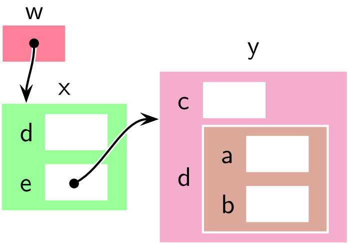
IR for Field Accesses Through Pointers: Example 2

```
struct A { double a; int b; };
struct B { int c; struct A d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = z;
int b = w->e->d.b;
```



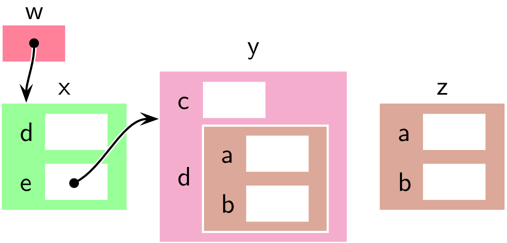
IR for Field Accesses Through Pointers: Example 2

```
struct A { double a; int b; };
struct B { int c; struct A d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = z;
int b = w->e->d.b;
```



IR for Field Accesses Through Pointers: Example 2

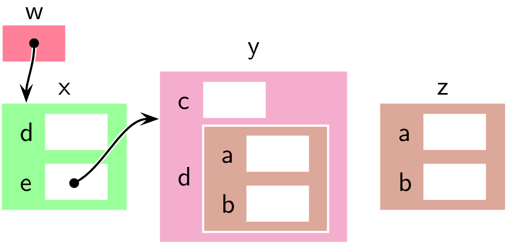
```
struct A { double a; int b; };
struct B { int c; struct A d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = z;
int b = w->e->d.b;
```



IR for Field Accesses Through Pointers: Example 2

```
struct A { double a; int b; };
struct B { int c; struct A d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = z;
int b = w->e->d.b;
```

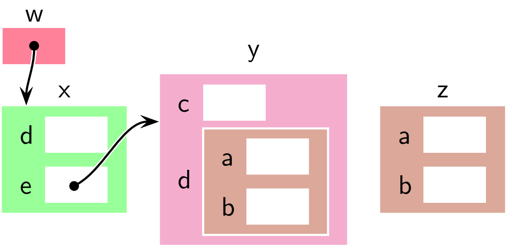
Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8



IR for Field Accesses Through Pointers: Example 2

```
struct A { double a; int b; };
struct B { int c; struct A d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = z;
int b = w->e->d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8



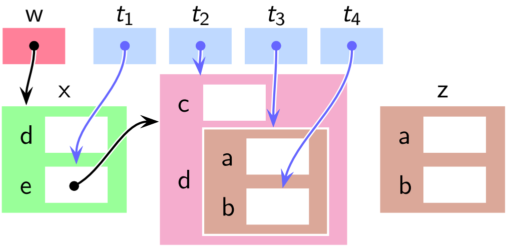
IR code for access expression `w->e->d.b`

$t_1 = w + 4 \quad //\&(x.e)$
$t_2 = *t_1 \quad //\&y$
$t_3 = t_2 + 4 \quad //\&(y.d)$
$t_4 = t_3 + 8 \quad //\&(y.d.b)$
$t_5 = *t_4$

IR for Field Accesses Through Pointers: Example 2

```
struct A { double a; int b; };
struct B { int c; struct A d; };
struct C { int d; struct B *e; };

struct C x, *w;
struct B y;
struct A z;

w = &x; x.e = &y; y.d = z;
int b = w->e->d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8



IR code for access expression `w->e->d.b`

$t_1 = w + 4 \quad //\&(x.e)$
$t_2 = *t_1 \quad //\&y$
$t_3 = t_2 + 4 \quad //\&(y.d)$
$t_4 = t_3 + 8 \quad //\&(y.d.b)$
$t_5 = *t_4$

SDD For Generating Code for Field Accesses Through Pointers

We use the following attributes

- E represents an arithmetic expression and F represents an access expression
- $E.place$, $E.code$, $id.name$, and $id.type$
- $F.type$: type of the field accessed using F
- $F.address$: name of the variable holding the address computed by F
- $F.code$: code representing the access expression F

  $pointer(\tau)$ denotes the type of a pointer to type τ

Unlike the previous approach, we cannot compute the final offsets at compile time because of pointers, and hence we use $F.code$ and not $F.offset$

We use the following functions apart from $gen(\cdot)$ and $getNewTemp(\,)$ functions

- $offset(\tau, f)$ gives the offset of field f in structure type τ
- $type(\tau, f)$ gives the type of field f in structure type τ

Grammar for Accessing Field Accesses Through Pointers

Since we need to use $\rightarrow$ as a token in our rules, we use quotes around it (i.e., '$\rightarrow$') to distinguish it from the metacharacter $\rightarrow$ that separates the LHS and RHS in the rule

$$
\begin{aligned}
E &\rightarrow F \\
F &\rightarrow id \cdot id \\
F &\rightarrow F \cdot id \\
F &\rightarrow id\ '{\rightarrow}'\ id \\
F &\rightarrow F\ '{\rightarrow}'\ id
\end{aligned}
$$

SDD for Generating IR for Field Accesses Through Pointers

$E \rightarrow F$	
$F \rightarrow id_1 \cdot id_2$	
$F_1 \rightarrow F_2 \cdot id$	

SDD for Generating IR for Field Accesses Through Pointers

$E \rightarrow F$	$t_1 = getNewTemp(\,); E.place = t_1$ $E.code = F.code \mid\mid gen(t_1, =, *F.address)$
$F \rightarrow id_1 \cdot id_2$	
$F_1 \rightarrow F_2 \cdot id$	

SDD for Generating IR for Field Accesses Through Pointers

$E \rightarrow F$	$t_1 = getNewTemp(\,);\ E.place = t_1$ $E.code = F.code \parallel gen(t_1, =, *F.address)$
$F \rightarrow id_1 \cdot id_2$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $F.type = pointer(type(id_1.type, id_2.name))$ $c_1 = gen(t_1, =, \&id_1.name)$ $F.code = c_1 \parallel gen(t_2, =, t_1 + offset(id_1.type, id_2.name))$ $F.address = t_2$
$F_1 \rightarrow F_2 \cdot id$	

SDD for Generating IR for Field Accesses Through Pointers

$E \rightarrow F$	$t_1 = getNewTemp(\,);\ E.place = t_1$ $E.code = F.code \,\|\|\, gen(t_1, =, *F.address)$
$F \rightarrow id_1 \cdot id_2$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $F.type = pointer(type(id_1.type, id_2.name))$ $c_1 = gen(t_1, =, \&id_1.name)$ $F.code = c_1 \,\|\|\, gen(t_2, =, t_1 + offset(id_1.type, id_2.name))$ $F.address = t_2$
$F_1 \rightarrow F_2 \cdot id$	$t_1 = getNewTemp(\,)$ $F_1.type = pointer(type(F_2.type, id.name))$ $c_1 = gen(t_1, =, F_2.address + offset(F_2.type, id.name))$ $F_1.code = F_2.code \,\|\|\, c_1$ $F_1.address = t_1$

IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Semantic Analysis

Section:

The Role of Semantic Analysis

Examples of Errors

Syntax Directed Definitions

Generating IR

Syntax Directed Translation Schemes

Type Analysis

Name and Scope Analysis

Declaration Processing

SDD For Generating IR for Field Accesses Through Pointers

$F \rightarrow id_1 \ '\!\rightarrow\!' \ id_2$	
$F_1 \rightarrow F_2 \ '\!\rightarrow\!' \ id$	

SDD For Generating IR for Field Accesses Through Pointers

$F \rightarrow id_1 \ ' \rightarrow ' \ id_2$	Let τ be a type such that $id_1.type = pointer(\tau)$ $t_1 = getNewTemp(\,)$ $F.type = pointer(type(\tau, id_2.name))$ $F.code = gen(t_1, =, id_1.name + offset(\tau, id_2.name))$ $F.address = t_1$
$F_1 \rightarrow F_2 \ ' \rightarrow ' \ id$	

SDD For Generating IR for Field Accesses Through Pointers

$F \rightarrow id_1 \ ' {\rightarrow} ' \ id_2$	Let τ be a type such that $id_1.type = pointer(\tau)$ $t_1 = getNewTemp(\,)$ $F.type = pointer(type(\tau, id_2.name))$ $F.code = gen(t_1, =, id_1.name + offset(\tau, id_2.name))$ $F.address = t_1$
$F_1 \rightarrow F_2 \ ' {\rightarrow} ' \ id$	Let τ be a type such that $F_2.type = pointer(\tau)$ $t_1 = getNewTemp(\,); t_2 = getNewTemp(\,)$ $F_1.type = pointer(type(\tau, id.name))$ $c_1 = gen(t_1, =, *F_2.address)$ $F_1.code = F_2.code \mid\mid c_1 \mid\mid gen(t_2, =, t_1 + offset(\tau, id.name))$ $F_1.address = t_2$

IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Semantic Analysis

Section:

The Role of Semantic Analysis

Examples of Errors

Syntax Directed Definitions

Generating IR

Syntax Directed Translation Schemes

Type Analysis

Name and Scope Analysis

Declaration Processing

Comparing the Rules for the Base Case

$F \rightarrow id_1 \cdot id_2$	
$F \rightarrow id_1 \; ' {\rightarrow} ' \; id_2$	

Comparing the Rules for the Base Case

$F \rightarrow id_1 \cdot id_2$	$t_1 = getNewTemp(\,);\ t_2 = getNewTemp(\,)$ $F.type = pointer(type(id_1.type, id_2.name))$ $c_1 = gen(t_1, =, \&id_1.name)$ $F.code = c_1 \mid\mid gen(t_2, =, t_1 + offset(id_1.type, id_2.name))$ $F.address = t_2$
$F \rightarrow id_1\ '\!\rightarrow\!'\ id_2$	Let τ be a type such that $id_1.type = pointer(\tau)$ $t_1 = getNewTemp(\,)$ $F.type = pointer(type(\tau, id_2.name))$ $F.code = gen(t_1, =, id_1.name + offset(\tau, id_2.name))$ $F.address = t_1$ name attribute me hi (*id) hota h

Note that we do not use the type of id_2

Comparing the Rules for the Recursive Case

$F_1 \rightarrow F_2 \cdot id$	
$F_1 \rightarrow F_2 \, '\!\rightarrow\!' \; id$	

Comparing the Rules for the Recursive Case

$F_1 \rightarrow F_2 \cdot id$	$t_1 = getNewTemp(\,)$ $F_1.type = pointer(type(F_2.type, id.name))$ $c_1 = gen(t_1, =, F_2.address + offset(F_2.type, id.name))$ $F_1.code = F_2.code \mid\mid c_1$ $F_1.address = t_1$
$F_1 \rightarrow F_2 \ '\!\rightarrow\!' \ id$	Let τ be a type such that $F_2.type = pointer(\tau)$ $t_1 = getNewTemp(\,); t_2 = getNewTemp(\,)$ $F_1.type = pointer(type(\tau, id.name))$ $c_1 = gen(t_1, =, *F_2.address)$ $F_1.code = F_2.code \mid\mid c_1 \mid\mid gen(t_2, =, t_1 + offset(\tau, id.name))$ $F_1.address = t_2$

Note that we do not use the type of *id*

Code for Field Accesses Through Pointers (Example 1)

Field Access

```
w->e->d->b;
```

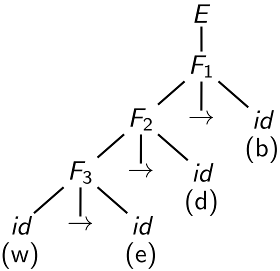
Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8

Code for Field Accesses Through Pointers (Example 1)

Field Access

```
w->e->d->b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8

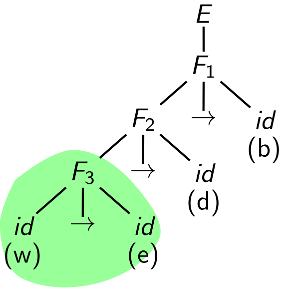


Code for Field Accesses Through Pointers (Example 1)

Field Access

```
w->e->d->b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8



$F_3.type$	struct B $**$
$F_3.code$	$t_1 = w + 4$
$F_3.address$	t_1

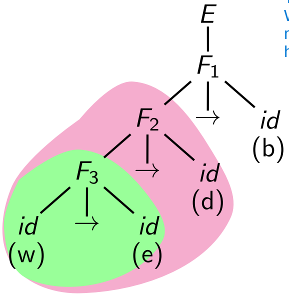
Code for Field Accesses Through Pointers (Example 1)

Field Access

```
w->e->d->b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8

w is a pointer to a struct C (not shown in table but implied)
e is a field in struct C that's a pointer to struct B (struct B*)
When we access w->e, we get a struct B*
F3 represents the evaluation of w->e
So F3.type is struct B** because:



The expression w->e yields a value of type struct B*
When generating the IR for this access, the compiler needs the address of this pointer (to later dereference it), hence the additional *

$F_3.type$	struct B $**$
$F_3.code$	$t_1 = w + 4$
$F_3.address$	t_1

$F_2.type$	struct A $**$
$F_2.code$	$t_1 = w + 4$ $t_2 = *t_1$ $t_3 = t_2 + 4$
$F_2.address$	t_3

Code for Field Accesses Through Pointers (Example 1)

We generate only t1 = w + 4 for F3 (not F2) because this IR uses a three-address code design with separate address computation and dereferencing operations.
The IR breaks the expression w->e->d->b into distinct steps:

Type	Field	Field Type	Offset
	d	int	0

Field A

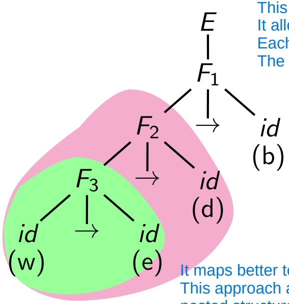
`w->e-`

Why do we generate a single statement $t_1 = w + 4$ for F_2 and not the sequence $t_1 = w + 4; t_2 = *t_1$?

Answered shortly

First, F3 calculates just the address of field e (t1 = w + 4), but doesn't dereference it yet
Then F2 performs the dereferencing (t2 = *t1) and continues to the next level

This separation is deliberate because:
It allows for better optimization opportunities
Each operation handles one specific task (address calculation or dereferencing)
The dereferencing is delayed until it's actually needed in the next access step



$F_3.type$	struct B $**$
$F_3.code$	$t_1 = w + 4$
$F_3.address$	t_1

$F_2.type$	struct A $**$
$F_2.code$	$t_1 = w + 4$ $t_2 = *t_1$ $t_3 = t_2 + 4$
$F_2.address$	t_3

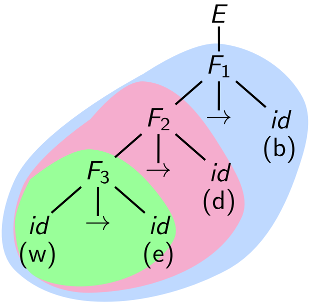
It maps better to actual machine instructions that may use different addressing modes
This approach also makes it easier to handle memory access patterns efficiently, especially with complex nested structures and pointer chains.

Code for Field Accesses Through Pointers (Example 1)

Field Access

```
w->e->d->b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8



$F_2.type$ struct A $**$

$F_2.code$

$$\begin{aligned} t_1 &= w + 4 \\ t_2 &= *t_1 \\ t_3 &= t_2 + 4 \end{aligned}$$

$F_2.address$ t_3

$F_1.type$ $int*$

$F_1.code$

$$\begin{aligned} t_1 &= w + 4 \\ t_2 &= *t_1 \\ t_3 &= t_2 + 4 \\ t_4 &= *t_3 \\ t_5 &= t_4 + 8 \end{aligned}$$

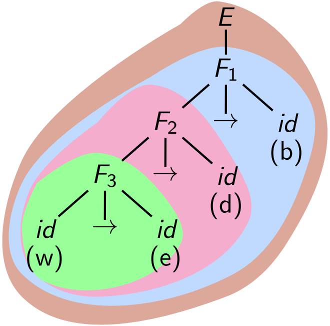
$F_1.address$ t_5

Code for Field Accesses Through Pointers (Example 1)

Field Access

```
w->e->d->b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A $*$	4
struct A	a	double	0
	b	int	8



$F_1.type$ $int*$

$F_1.code$

$$\begin{aligned} t_1 &= w + 4 \\ t_2 &= *t_1 \\ t_3 &= t_2 + 4 \\ t_4 &= *t_3 \\ t_5 &= t_4 + 8 \end{aligned}$$

$F_1.address$ t_5

$E.code$

$$\begin{aligned} t_1 &= w + 4 \\ t_2 &= *t_1 \\ t_3 &= t_2 + 4 \\ t_4 &= *t_3 \\ t_5 &= t_4 + 8 \\ t_6 &= *t_5 \end{aligned}$$

$E.place$ t_6

What does F Represent?

In productions $F \rightarrow id_1\ \ '\rightarrow'\ \ id$, $F \rightarrow F_1\ \ '\rightarrow'\ \ id$, $F \rightarrow id_1 \cdot id$, and $F \rightarrow F_1 \cdot id$, non-terminal F (occurring on the LHS) represents the field named $id.name$. We want $F.address$ to represent a pointer to this field. There are three possibilities for this field:

- It is a structure variable whose field is accessed further.
  In this case, we add the offset of the further field to $F.address$.
- It is a pointer to a structure variable whose field is accessed further.
  In this case, we add dereference $F.address$ and the offset of the further field to it.
- In all other cases, we dereference $F.address$.

This decision depends on the type of id in the two productions which is not checked by our semantic rules; they check the type of id_1 and F_1 in the productions above.

Hence this decision is left for the occurrence of F in the RHS of the productions.

Code for Field Accesses Through Pointers (Example 2)

Field Access

```
w->e->d.b;
```

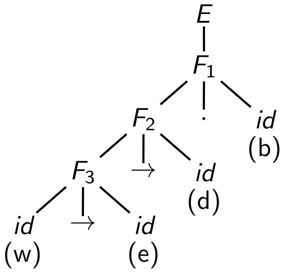
Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8

Code for Field Accesses Through Pointers (Example 2)

Field Access

```
w->e->d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8

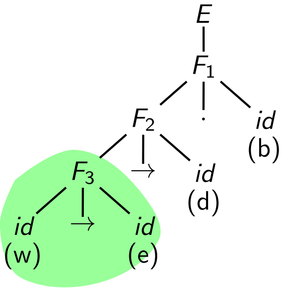


Code for Field Accesses Through Pointers (Example 2)

Field Access

```
w->e->d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8



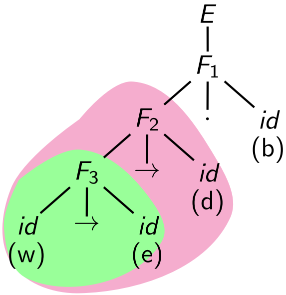
$F_3.type$	struct B $**$
$F_3.code$	$t_1 = w + 4$
$F_3.address$	t_1

Code for Field Accesses Through Pointers (Example 2)

Field Access

```
w->e->d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8



F3 is structure variable here

$F_3.type$ struct B $**$

$F_3.code$ $t_1 = w + 4$

$F_3.address$ t_1

F2 is pointer to structure variable here

$F_2.type$ struct A $**$

$F_2.code$ $t_1 = w + 4$; $t_2 = *t_1$; $t_3 = t_2 + 4$

$F_2.address$ t_3

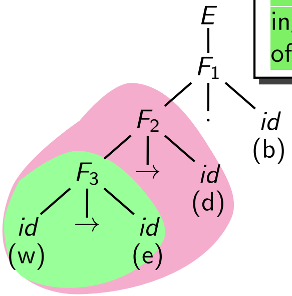
Code for Field Accesses Through Pointers (Example 2)

Field Access

```
w->e->d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
			8

The next field access operator is $\cdot$ and hence instead of dereferencing t_3 in $F_2.code$ (or $F_1.code$), the offset of b should be added to it



$F_3.type$	struct B $**$
$F_3.code$	$t_1 = w + 4$
$F_3.address$	t_1

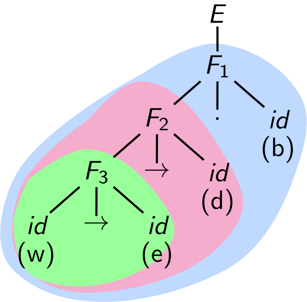
$F_2.type$	struct A $**$
$F_2.code$	$t_1 = w + 4$ $t_2 = *t_1$ $t_3 = t_2 + 4$
$F_2.address$	t_3

Code for Field Accesses Through Pointers (Example 2)

Field Access

```
w->e->d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8



$F_2.type$ struct A $**$

$F_2.code$

$$t_1 = w + 4$$
$$t_2 = *t_1$$
$$t_3 = t_2 + 4$$

$F_2.address$ t_3

$F_1.type$ $int*$

$F_1.code$

$$t_1 = w + 4$$
$$t_2 = *t_1$$
$$t_3 = t_2 + 4$$
$$t_4 = t_3 + 8$$

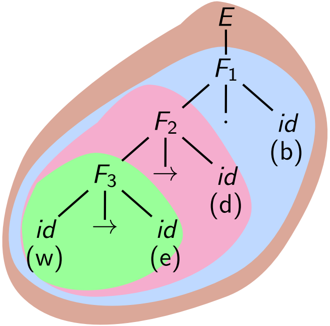
$F_1.address$ t_4

Code for Field Accesses Through Pointers (Example 2)

Field Access

```
w->e->d.b;
```

Type	Field	Field Type	Offset
struct C	d	int	0
	e	struct B $*$	4
struct B	c	int	0
	d	struct A	4
struct A	a	double	0
	b	int	8



$F_1.type$ $int*$

$F_1.code$
$$t_1 = w + 4$$
$$t_2 = *t_1$$
$$t_3 = t_2 + 4$$
$$t_4 = t_3 + 8$$

$F_1.address$ t_4

$E.code$
$$t_1 = w + 4$$
$$t_2 = *t_1$$
$$t_3 = t_2 + 4$$
$$t_4 = t_3 + 8$$
$$t_5 = *t_4$$

$E.place$ t_5

The Role of Semantic Analysis
Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

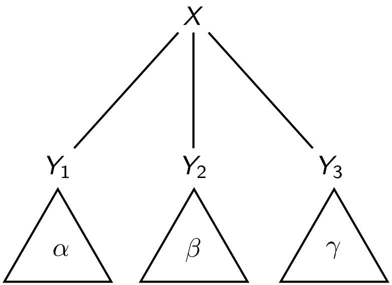
MIDSEM SYLLABUS UPTO HERE

Syntax Directed Translation Schemes

Inherited and Synthesized Attributes

Given a production $X \rightarrow Y_1 Y_2 \ldots Y_k$

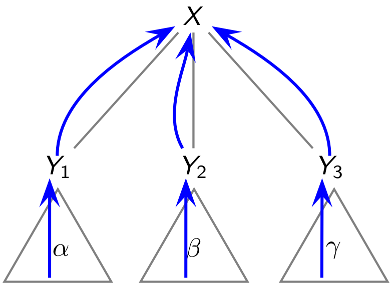
- If an attribute $X.a$ is computed from those of Y_i, $1 \leq i \leq k$, the $X.a$ is a synthesized attribute
- If an attribute $Y_i.a$, $1 \leq i \leq k$ is computed from from those of X or Y_i, $1 \leq i \leq k$, then $Y_i.a$ is an inherited attribute



Inherited and Synthesized Attributes

Given a production $X \rightarrow Y_1 Y_2 \ldots Y_k$

- If an attribute $X.a$ is computed from those of Y_i, $1 \leq i \leq k$, the $X.a$ is a synthesized attribute
- If an attribute $Y_i.a$, $1 \leq i \leq k$ is computed from from those of X or Y_i, $1 \leq i \leq k$, then $Y_i.a$ is an inherited attribute

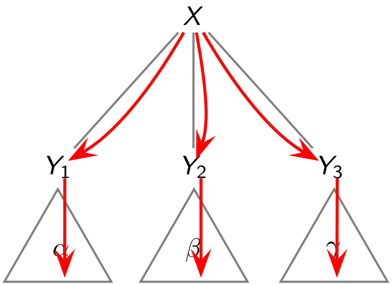


Synthesized attributes (blue arrows) flow upwards in a parse tree (computed from descendants)

Inherited and Synthesized Attributes

Given a production $X \rightarrow Y_1 Y_2 \dots Y_k$

- If an attribute $X.a$ is computed from those of Y_i, $1 \leq i \leq k$, the $X.a$ is a synthesized attribute
- If an attribute $Y_i.a$, $1 \leq i \leq k$ is computed from from those of X or Y_i, $1 \leq i \leq k$, then $Y_i.a$ is an inherited attribute



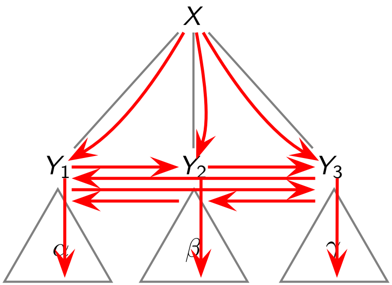
Synthesized attributes (blue arrows) flow upwards in a parse tree (computed from descendants)

Inherited attributes (red arrows) flow downwards or sideways in a parse tree (computed from ancestors or siblings)

Inherited and Synthesized Attributes

Given a production $X \rightarrow Y_1 Y_2 \ldots Y_k$

- If an attribute $X.a$ is computed from those of Y_i, $1 \leq i \leq k$, the $X.a$ is a synthesized attribute
- If an attribute $Y_i.a$, $1 \leq i \leq k$ is computed from from those of X or Y_i, $1 \leq i \leq k$, then $Y_i.a$ is an inherited attribute



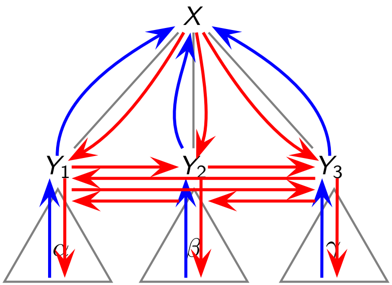
Synthesized attributes (blue arrows) flow upwards in a parse tree (computed from descendants)

Inherited attributes (red arrows) flow downwards or sideways in a parse tree (computed from ancestors or siblings)

Inherited and Synthesized Attributes

Given a production $X \rightarrow Y_1 Y_2 \ldots Y_k$

- If an attribute $X.a$ is computed from those of Y_i, $1 \le i \le k$, the $X.a$ is a synthesized attribute
- If an attribute $Y_i.a$, $1 \le i \le k$ is computed from from those of X or Y_i, $1 \le i \le k$, then $Y_i.a$ is an inherited attribute



Synthesized attributes (blue arrows) flow upwards in a parse tree (computed from descendants)

Inherited attributes (red arrows) flow downwards or sideways in a parse tree (computed from ancestors or siblings)

Why Inherited Attributes?

Consider an SDD for processing declarations

$Decl \rightarrow Type\ \ VarList$	$VarList.type = Type.name$
$Type \rightarrow \text{int}$	$Type.name = \text{int}$
$Type \rightarrow \text{float}$	$Type.name = \text{float}$
$VarList_1 \rightarrow VarList_2\ ,\ id$	$VarList_2.type = VarList_1.type$ $id.type = VarList_1.type$
$VarList \rightarrow id$	$id.type = VarList.type$

Here, the attribute *type* is inherited

Why Inherited Attributes?

Consider IR Generation for a **for** loop with **break** and **continue** statements

$S_1 \rightarrow$ **for** $(E_1;\ E_2;\ E_3)\ S_2$	$\ldots$
$S \rightarrow$ **break**	$S.code = gen(\text{goto}, S.exit)$
$S \rightarrow$ **continue**	$S.code = gen(\text{goto}, S.increment)$

We need the labels $S.exit$ and $S.increment$ while parsing the string derivable from S_2

We see later, how they are used

Control Flow Translation of Boolean Expressions

Short-circuit evaluation of boolean expressions	
$E_1 \rightarrow E_2$ **or** E_3	Evaluate E_3 only if E_2 evaluates to false because if E_2 evaluates to true, E_1 is true regardless of E_2
$E_1 \rightarrow E_2$ **and** E_3	Evaluate E_3 only if E_2 evaluates to true because if E_2 evaluates to false, E_1 is false regardless of E_2

Control Flow Translation of Boolean Expressions

Short-circuit evaluation of boolean expressions	
$E_1 \rightarrow E_2$ **or** E_3	Evaluate E_3 only if E_2 evaluates to false because if E_2 evaluates to true, E_1 is true regardless of E_2
$E_1 \rightarrow E_2$ **and** E_3	Evaluate E_3 only if E_2 evaluates to true because if E_2 evaluates to false, E_1 is false regardless of E_2

Input Expression	Generated Code
$(a < b$ **or** $b > d)$ **and** $c > d$	$t_1 = a < b$ if t_1 goto $L3$ goto $L4$ $L4: t_2 = b > c$ if t_2 goto $L3$ goto $L2$ $L3: t_3 = c > d$ if t_3 goto $L1$ // overall true goto $L2$ // overall false

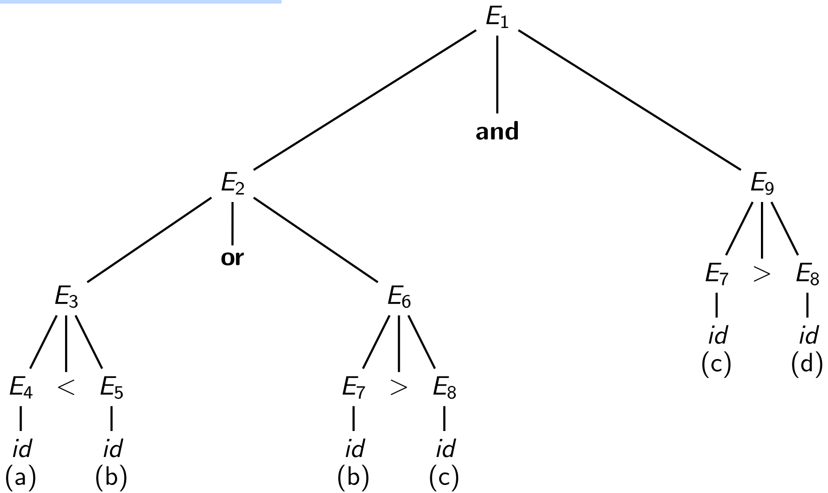
SDD for Control Flow Translation of Boolean Expressions

$E_1 \rightarrow E_2$ **or** E_3	$E_2.true = E_1.true$ $E_2.false = getNewLabel()$ $E_3.true = E_1.true$ $E_3.false = E_1.false$ $E_1.code = E_2.code \mid\mid gen(E_2.false, :) \mid\mid E_3.code$
$E_1 \rightarrow E_2$ **and** E_3	$E_2.true = getNewLabel()$ $E_2.false = E_1.false$ $E_3.true = E_1.true$ $E_3.false = E_1.false$ $E_1.code = E_2.code \mid\mid gen(E_2.true, :) \mid\mid E_3.code$
$E_1 \rightarrow E_2$ **relop** E_3	$t_1 = getNewTemp()$ $c_1 = gen(t_1, =, E_2.place, \text{relop}, E_3.place)$ $c_2 = gen(\text{if}, t_1, \text{goto}, E_1.true)$ $c_3 = gen(\text{goto}, E_1.false)$ $E_1.code = E_2.code \mid\mid E_3.code \mid\mid c_1 \mid\mid c_2 \mid\mid c_3$

Attribute Evaluation for Control Flow Translation of Boolean Expressions

Input Expression:

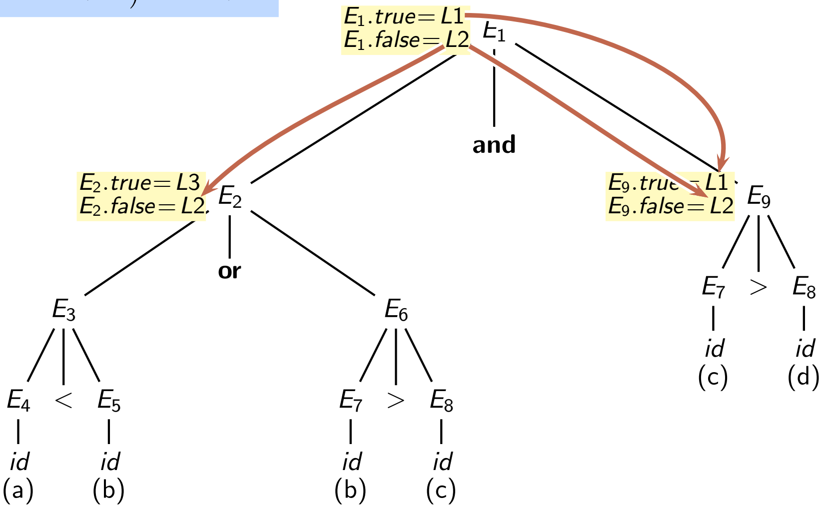
$\left(a < b\right.$ **or** $\left.b > d\right)$ **and** $c > d$



Attribute Evaluation for Control Flow Translation of Boolean Expressions

Input Expression:

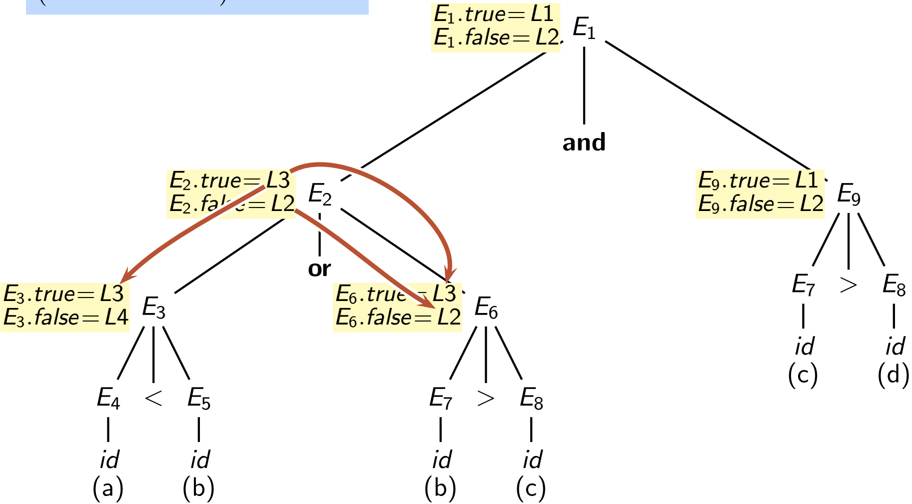
$(a < b \textbf{ or } b > d) \textbf{ and } c > d$



Attribute Evaluation for Control Flow Translation of Boolean Expressions

Input Expression:

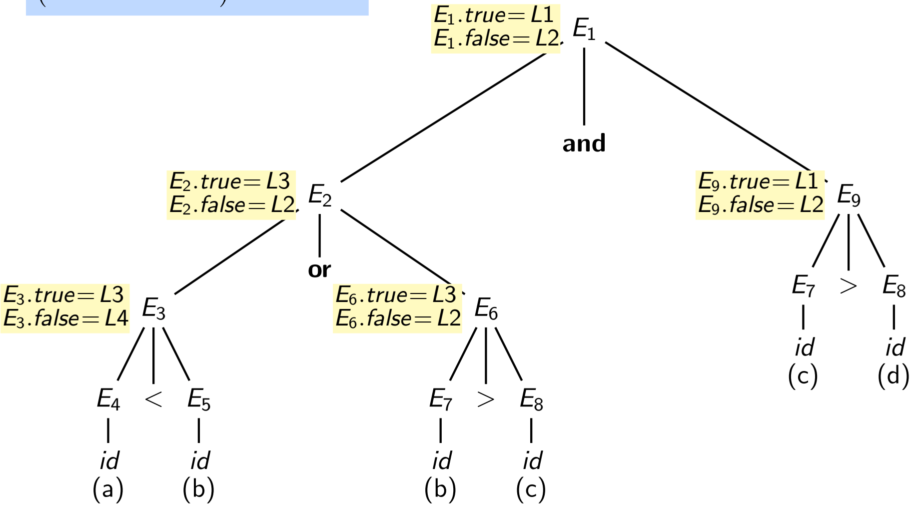
$(a < b$ **or** $b > d)$ **and** $c > d$



Attribute Evaluation for Control Flow Translation of Boolean Expressions

Input Expression:

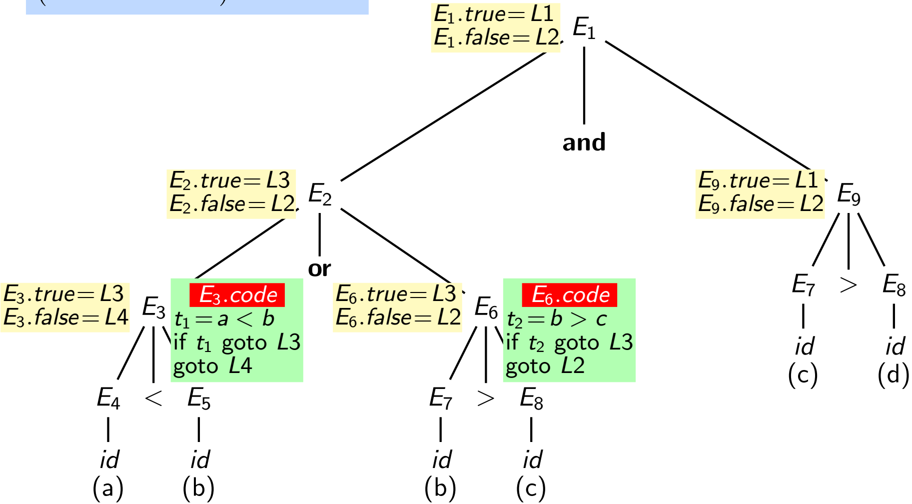
$(a < b \textbf{ or } b > d) \textbf{ and } c > d$



Attribute Evaluation for Control Flow Translation of Boolean Expressions

Input Expression:

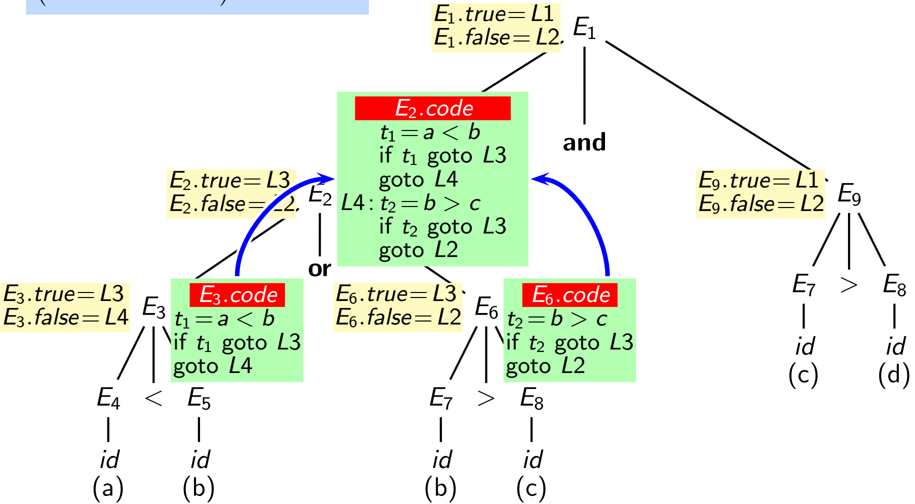
$(a < b$ **or** $b > d)$ **and** $c > d$



Attribute Evaluation for Control Flow Translation of Boolean Expressions

Input Expression:

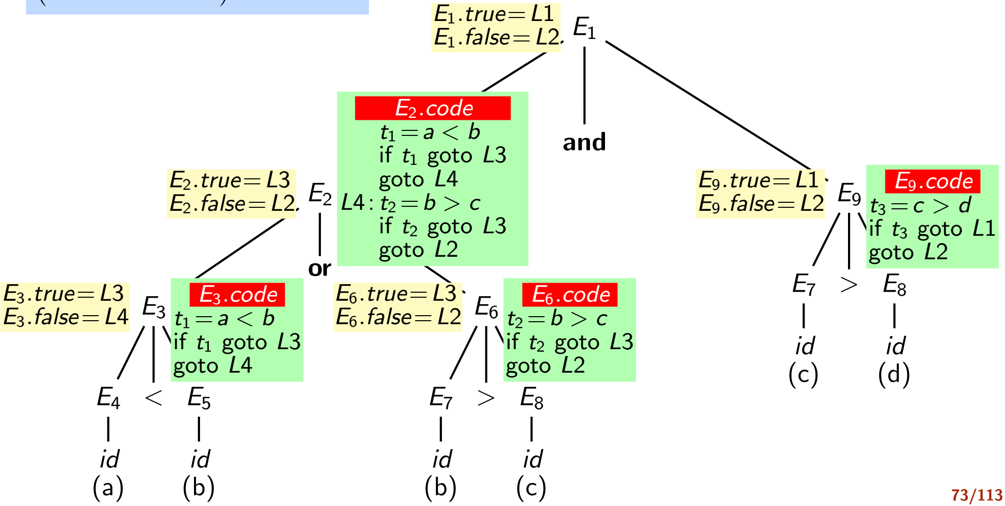
$\left(a < b \textbf{ or } b > d\right) \textbf{ and } c > d$



Attribute Evaluation for Control Flow Translation of Boolean Expressions

Input Expression:

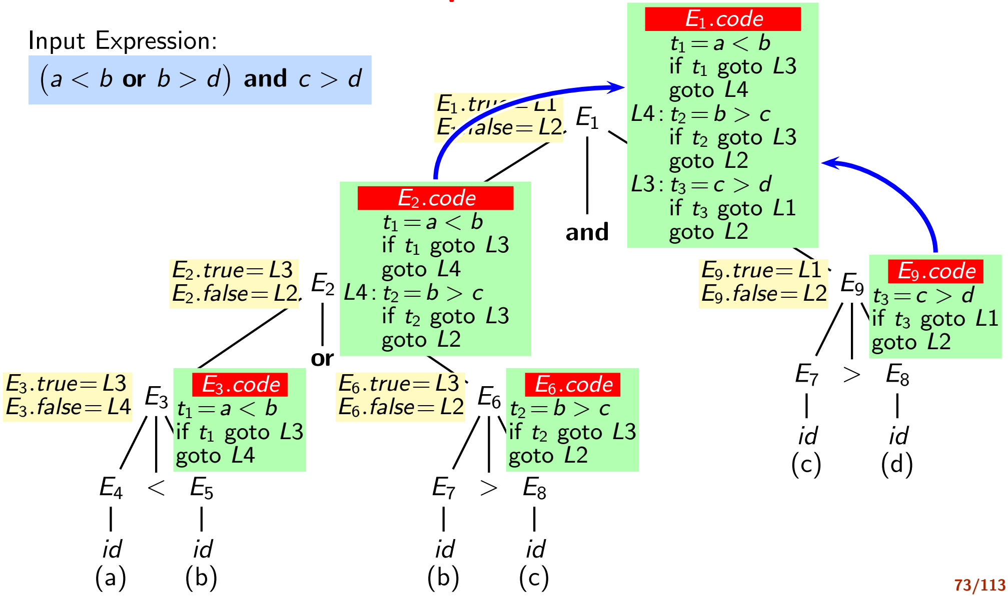
$(a < b$ **or** $b > d)$ **and** $c > d$



Attribute Evaluation for Control Flow Translation of Boolean Expressions

Input Expression:

$(a < b \textbf{ or } b > d) \textbf{ and } c > d$



IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Semantic Analysis

Section:

The Role of Semantic Analysis

Examples of Errors

Syntax Directed Definitions

Generating IR

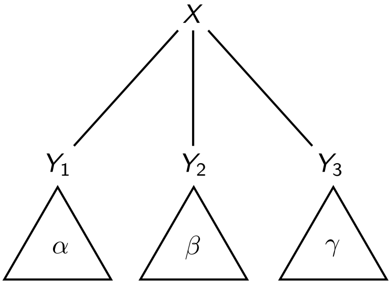
Syntax Directed Translation Schemes

Type Analysis

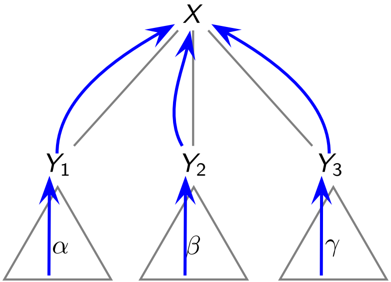
Name and Scope Analysis

Declaration Processing

Computing Inherited Attributes Concurrently with Parsing

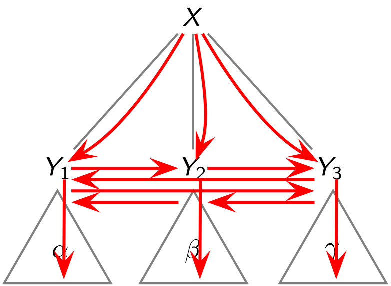


Computing Inherited Attributes Concurrently with Parsing



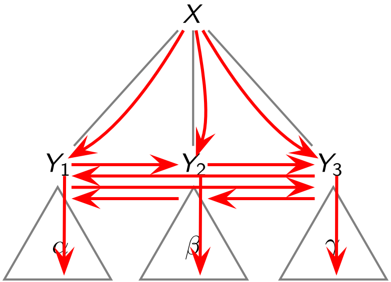
- Synthesized attributes can be easily computed during bottom-up parsing

Computing Inherited Attributes Concurrently with Parsing



- Synthesized attributes can be easily computed during bottom-up parsing
- Inherited attributes cannot be computed if they depend on a symbol not yet seen

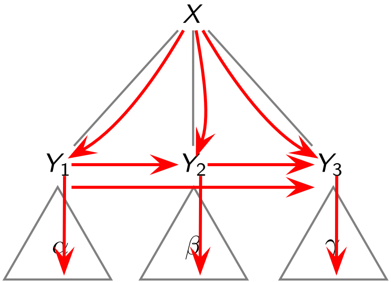
Computing Inherited Attributes Concurrently with Parsing



- Synthesized attributes can be easily computed during bottom-up parsing
- Inherited attributes cannot be computed if they depend on a symbol not yet seen
- We can restrict the inherited attributes to depend only on the attributes of grammar symbols that have been seen

  Given a production $X \rightarrow Y_1 Y_2 \ldots Y_k$

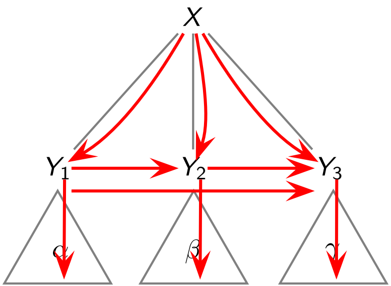
Computing Inherited Attributes Concurrently with Parsing



- Synthesized attributes can be easily computed during bottom-up parsing
- Inherited attributes cannot be computed if they depend on a symbol not yet seen
- We can restrict the inherited attributes to depend only on the attributes of grammar symbols that have been seen

  Given a production $X \rightarrow Y_1 Y_2 \ldots Y_k$
  - $Y_i.a$, is computed only from the attributes of X or Y_j, $j < i$

Computing Inherited Attributes Concurrently with Parsing



- Synthesized attributes can be easily computed during bottom-up parsing
- Inherited attributes cannot be computed if they depend on a symbol not yet seen
- We can restrict the inherited attributes to depend only on the attributes of grammar symbols that have been seen

  Given a production $X \rightarrow Y_1 Y_2 \ldots Y_k$
  - $Y_i.a$, is computed only from the attributes of X or Y_j, $j < i$
  - $X.a$ would have been computed from the grammar symbols that have already been seen
    (i.e., in some production $Z \rightarrow \alpha X \beta$)

S-Attributed and L-Attributed SDDs

- An SDD is *S-attributed* if it uses only synthesized attributes
- An SDD is *L-attributed* if it uses synthesized attributes or inherited attributes that depend on some symbol to the left
  - Given a production $X \rightarrow Y_1 Y_2 \ldots Y_k$ attribute $Y_i.a$, of some Y_i is computed only from the attributes of X or Y_j, $j < i$
  - Symbols X and Y_j, $j < i$ appear to the left of Y_i in the production

S-Attributed and L-Attributed SDDs

- An SDD is *S-attributed* if it uses only synthesized attributes
- An SDD is *L-attributed* if it uses synthesized attributes or inherited attributes that depend on some symbol to the left
  - Given a production $X \rightarrow Y_1 Y_2 \ldots Y_k$ attribute $Y_i.a$, of some Y_i is computed only from the attributes of X or Y_j, $j < i$
  - Symbols X and Y_j, $j < i$ appear to the left of Y_i in the production
- All SDDs in the previous section are S-attributed whereas the declaration processing SDD is L-attributed

$Decl \rightarrow Type \;\; VarList$	$VarList.type = \; Type.name$
$Type \rightarrow \text{int}$	$Type.name = \text{int}$
$Type \rightarrow \text{float}$	$Type.name = \text{float}$
$VarList_1 \rightarrow VarList_2 \; , \; id$	$VarList_2.type = VarList_1.type$ $id.type = VarList_1.type$
$VarList \rightarrow id$	$id.type = VarList.type$

Syntax Directed Translation Schemes (SDTS)

- A Syntax Directed Translation Scheme is an SDD with the following two changes
  - Semantic rules are replaced by actions possibly with side effects
    We include the actions in a pair of braces (i.e., within "{" and "}")
  - The exact time of the action is specified; an action computing an inherited attribute of a non-terminal appears just before the non-terminal
- The SDTS for declaration processing is as follows

$$
\begin{aligned}
Decl &\rightarrow Type\ \{VarList.type = Type.name\}\ VarList \\
Type &\rightarrow \text{int}\ \{Type.name = \text{int}\} \\
Type &\rightarrow \text{float}\ \{Type.name = \text{float}\} \\
VarList_1 &\rightarrow \{VarList_2.type = VarList_1.type\}\ VarList_2\ , \\
&\qquad id\ \{id.type = VarList_1.type\} \\
VarList &\rightarrow id\ \{id.type = VarList.type\}
\end{aligned}
$$

S-Attributed and L-Attributed SDTSs

- An S-Attributed SDTS uses only synthesized attributes and all actions appear at the end of the RHS of a production
- An L-Attributed SDTS uses synthesized attributes or attributes that depend on a symbol towards the left of the grammar symbols of the attributes

  The actions may appear in the middle of the rules or at the end of the RHS of a production
- The SDTS for declaration processing is L-attributed

How and when are the actions in the middle of a rule executed?

- A production with an action in the middle is transformed into two productions

$X \rightarrow Y_1 \ \{ \text{ action } \} \ Y_2$ $\Longrightarrow$ $\begin{array}{l} X \rightarrow Y_1 \ M \ Y_2 \\ M \rightarrow \epsilon \ \{ \text{ action } \} \end{array}$

  where M is a marker non-terminal for Y_2

- The action is executed after reduction by $M \rightarrow \epsilon$

  It is convenient to execute actions consistently after a reduction

- A distinct marker non-terminal is introduced for every such action

  We have as many additional ϵ-productions as the number of such actions

Representing the Actions in the Middle by Marker Non-Terminals

$$Decl \rightarrow Type\ \{VarList.type = Type.name\}\ VarList$$

$$Type \rightarrow \text{int}\ \{Type.name = \text{int}\}$$

$$Type \rightarrow \text{float}\ \{Type.name = \text{float}\}$$

$$VarList_1 \rightarrow \{VarList_2.type = VarList_1.type\}\ VarList_2\ ,$$

$$id\ \{id.type = VarList_1.type\}$$

$$VarList \rightarrow id\ \{id.type = VarList.type\}$$

Representing the Actions in the Middle by Marker Non-Terminals

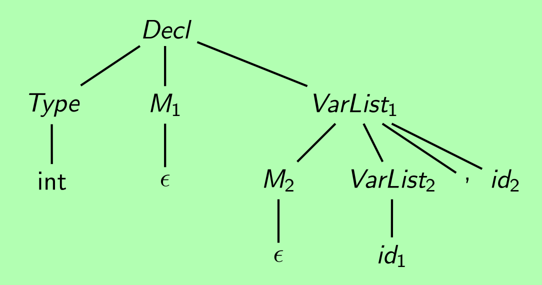
$$Decl \rightarrow Type\ \{VarList.type = Type.name\}\ VarList$$

$$Type \rightarrow \text{int}\ \{Type.name = \text{int}\}$$

$$Type \rightarrow \text{float}\ \{Type.name = \text{float}\}$$

$$VarList_1 \rightarrow \{VarList_2.type = VarList_1.type\}\ VarList_2\ ,\ id\ \{id.type = VarList_1.type\}$$

$$VarList \rightarrow id\ \{id.type = VarList.type\}$$



Attribute Evaluation

Representing the Actions in the Middle by Marker Non-Terminals

$$Decl \rightarrow Type \ \{VarList.type = Type.name\} \ VarList$$

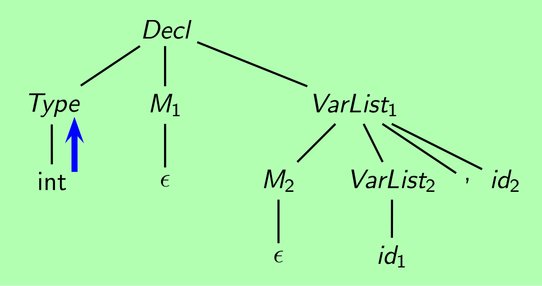
$$Type \rightarrow \text{int} \ \{Type.name = \text{int}\}$$

$$Type \rightarrow \text{float} \ \{Type.name = \text{float}\}$$

$$VarList_1 \rightarrow \{VarList_2.type = VarList_1.type\} \ VarList_2 \ ,$$

$$id \ \{id.type = VarList_1.type\}$$

$$VarList \rightarrow id \ \{id.type = VarList.type\}$$



Attribute Evaluation

$Type.name = \text{int}$

Representing the Actions in the Middle by Marker Non-Terminals

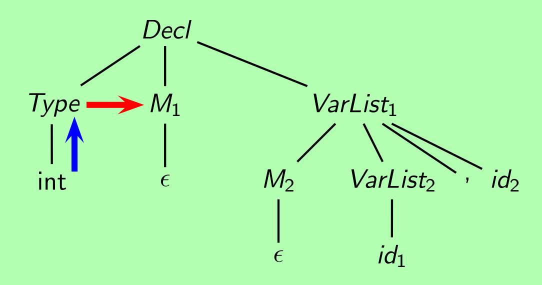
$$Decl \rightarrow Type \ \{VarList.type = Type.name\} \ VarList$$

$$Type \rightarrow \text{int} \ \{Type.name = \text{int}\}$$

$$Type \rightarrow \text{float} \ \{Type.name = \text{float}\}$$

$$VarList_1 \rightarrow \{VarList_2.type = VarList_1.type\} \ VarList_2 \ , \ id \ \{id.type = VarList_1.type\}$$

$$VarList \rightarrow id \ \{id.type = VarList.type\}$$



Attribute Evaluation

$Type.name = \text{int}$

$VarList_1.type = \text{int}$

Representing the Actions in the Middle by Marker Non-Terminals

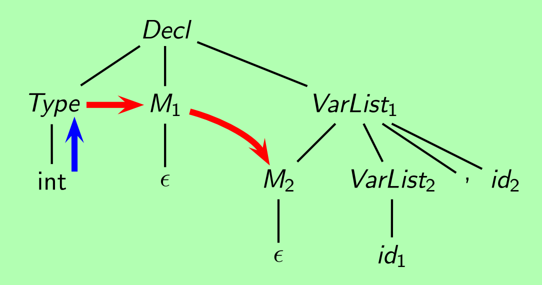
$$Decl \rightarrow Type\ \{VarList.type = Type.name\}\ VarList$$

$$Type \rightarrow \text{int}\ \{Type.name = \text{int}\}$$

$$Type \rightarrow \text{float}\ \{Type.name = \text{float}\}$$

$$VarList_1 \rightarrow \{VarList_2.type = VarList_1.type\}\ VarList_2\ ,\ id\ \{id.type = VarList_1.type\}$$

$$VarList \rightarrow id\ \{id.type = VarList.type\}$$



Attribute Evaluation

$Type.name = \text{int}$

$VarList_1.type = \text{int}$

$VarList_2.type = \text{int}$

Representing the Actions in the Middle by Marker Non-Terminals

$$Decl \rightarrow Type\ \{VarList.type = Type.name\}\ VarList$$

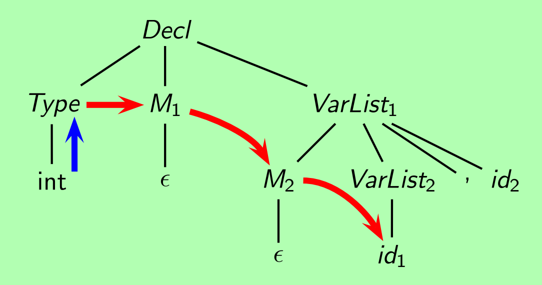
$$Type \rightarrow \text{int}\ \{Type.name = \text{int}\}$$

$$Type \rightarrow \text{float}\ \{Type.name = \text{float}\}$$

$$VarList_1 \rightarrow \{VarList_2.type = VarList_1.type\}\ VarList_2\ ,$$

$$id\ \{id.type = VarList_1.type\}$$

$$VarList \rightarrow id\ \{id.type = VarList.type\}$$



Attribute Evaluation

$Type.name = \text{int}$

$VarList_1.type = \text{int}$

$VarList_2.type = \text{int}$

$id_1.type = \text{int}$

Representing the Actions in the Middle by Marker Non-Terminals

$$Decl \rightarrow Type \ \{VarList.type = Type.name\} \ VarList$$

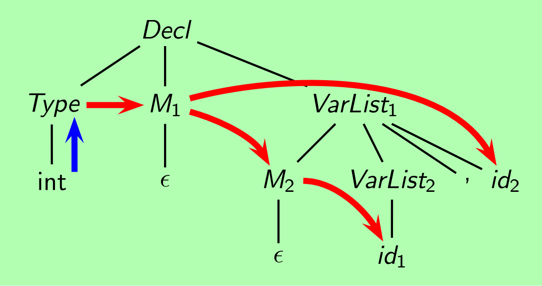
$$Type \rightarrow \text{int} \ \{Type.name = \text{int}\}$$

$$Type \rightarrow \text{float} \ \{Type.name = \text{float}\}$$

$$VarList_1 \rightarrow \{VarList_2.type = VarList_1.type\} \ VarList_2 \ ,$$

$$id \ \{id.type = VarList_1.type\}$$

$$VarList \rightarrow id \ \{id.type = VarList.type\}$$



Attribute Evaluation

$Type.name = \text{int}$

$VarList_1.type = \text{int}$

$VarList_2.type = \text{int}$

$id_1.type = \text{int}$

$id_2.type = \text{int}$

The Role of Marker Non-Terminals

- Marker non-terminals facilitate a corresponding slot on the value stack where the inherited attribute of the next grammar symbol can be stored
- Marker non-terminals may introduce reduce-reduce conflicts because of the ϵ rules

Marker Non-Terminals Facilitate Recording Inherited Attributes

M is a marker non-terminal for Y_2 in the grammar on the right

$Y_1.s$ and $Y_2.s$ denote the synthesized attributes of Y_1 and Y_2 whereas $Y_2.i$ denotes the inherited attribute of Y_2

$$\begin{aligned} X &\rightarrow Y_1\ M\ Y_2 \\ M &\rightarrow \epsilon\ \{\ldots\} \\ Y_2 &\rightarrow \alpha\ \{\ldots\} \end{aligned}$$

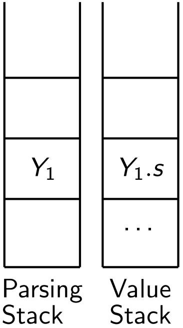
Marker Non-Terminals Facilitate Recording Inherited Attributes

M is a marker non-terminal for Y_2 in the grammar on the right

$Y_1.s$ and $Y_2.s$ denote the synthesized attributes of Y_1 and Y_2 whereas $Y_2.i$ denotes the inherited attribute of Y_2

$$X \to Y_1\ M\ Y_2$$
$$M \to \epsilon\ \{\ldots\}$$
$$Y_2 \to \alpha\ \{\ldots\}$$

Before reducing by $M \to \epsilon\ \{\ \ldots\ \}$



The Role of Semantic Analysis
Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

Marker Non-Terminals Facilitate Recording Inherited Attributes

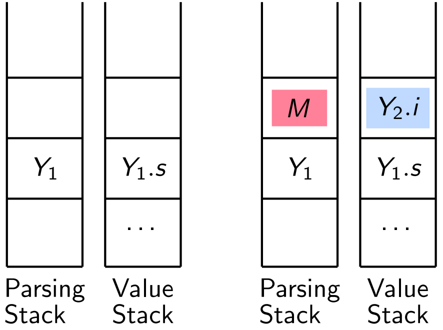
M is a marker non-terminal for Y_2 in the grammar on the right

$Y_1.s$ and $Y_2.s$ denote the synthesized attributes of Y_1 and Y_2 whereas $Y_2.i$ denotes the inherited attribute of Y_2

$$X \rightarrow Y_1\ M\ Y_2$$
$$M \rightarrow \epsilon\ \{\ldots\}$$
$$Y_2 \rightarrow \alpha\ \{\ldots\}$$

Before reducing by $M \rightarrow \epsilon\ \{\ \ldots\ \}$

After reducing by $M \rightarrow \epsilon\ \{\ \ldots\ \}$



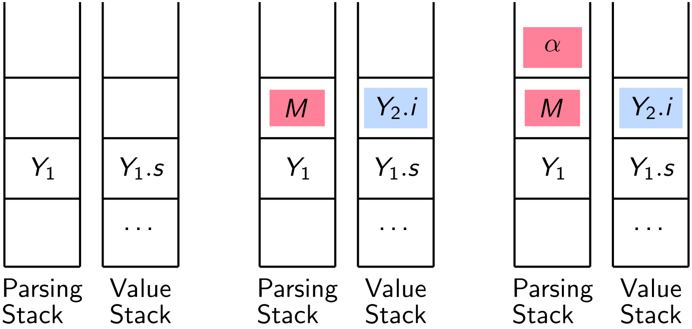
Marker Non-Terminals Facilitate Recording Inherited Attributes

M is a marker non-terminal for Y_2 in the grammar on the right

$Y_1.s$ and $Y_2.s$ denote the synthesized attributes of Y_1 and Y_2 whereas $Y_2.i$ denotes the inherited attribute of Y_2

$$X \rightarrow Y_1\ M\ Y_2$$
$$M \rightarrow \epsilon\ \{\ldots\}$$
$$Y_2 \rightarrow \alpha\ \{\ldots\}$$

Before reducing by $M \rightarrow \epsilon\ \{\ \ldots\ \}$	After reducing by $M \rightarrow \epsilon\ \{\ \ldots\ \}$	After pushing handle α



Marker Non-Terminals Facilitate Recording Inherited Attributes

M is a marker non-terminal for Y_2 in the grammar on the right

$Y_1.s$ and $Y_2.s$ denote the synthesized attributes of Y_1 and Y_2 whereas $Y_2.i$ denotes the inherited attribute of Y_2

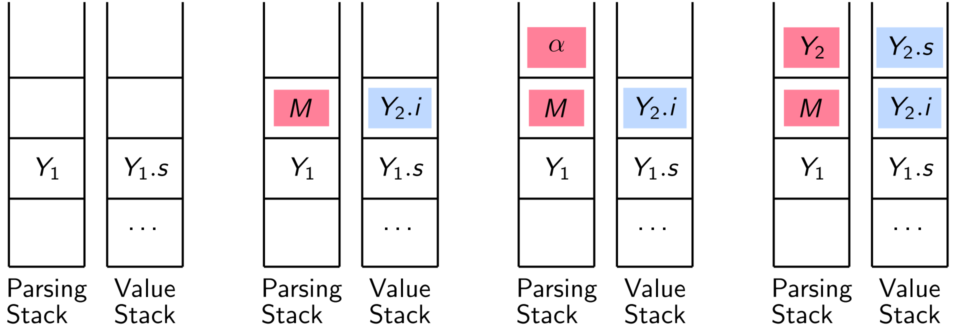
$$X \to Y_1 \; M \; Y_2$$
$$M \to \epsilon \; \{\ldots\}$$
$$Y_2 \to \alpha \; \{\ldots\}$$

Before reducing by $M \to \epsilon \; \{ \ldots \}$

After reducing by $M \to \epsilon \; \{ \ldots \}$

After pushing handle α

After reducing by $Y_2 \to \alpha \; \{ \ldots \}$



Marker Non-Terminals May Cause Reduce-Reduce Conflicts

Consider the grammar of declaration consisting of non-terminals D (Declaration), T (Type), L (List of identifiers), terminals int, "," and id, and marker non-terminals M_1, M_2, and M_3

$$
\begin{aligned}
D &\rightarrow T\ M_1\ L \\
T &\rightarrow \text{int} \\
L &\rightarrow M_2\ L\ ,\ id \\
L &\rightarrow M_3\ id \\
M_1 &\rightarrow \epsilon \\
M_2 &\rightarrow \epsilon \\
M_3 &\rightarrow \epsilon
\end{aligned}
$$

Marker Non-Terminals May Cause Reduce-Reduce Conflicts

Consider the grammar of declaration consisting of non-terminals D (Declaration), T (Type), L (List of identifiers), terminals int, "," and id, and marker non-terminals M_1, M_2, and M_3

$$
\begin{aligned}
D &\rightarrow T\ M_1\ L \\
T &\rightarrow \text{int} \\
L &\rightarrow M_2\ L\ ,\ id \\
L &\rightarrow M_3\ id \\
M_1 &\rightarrow \epsilon \\
M_2 &\rightarrow \epsilon \\
M_3 &\rightarrow \epsilon
\end{aligned}
$$

I_0

$$
\boxed{
\begin{aligned}
D' &\rightarrow \bullet\, D \\
D &\rightarrow \bullet\, T\ M_1\ L \\
T &\rightarrow \bullet\, \text{int}
\end{aligned}
}
$$

Marker Non-Terminals May Cause Reduce-Reduce Conflicts

Consider the grammar of declaration consisting of non-terminals D (Declaration), T (Type), L (List of identifiers), terminals int, "," and id, and marker non-terminals M_1, M_2, and M_3

$$
\begin{aligned}
D &\rightarrow T\ M_1\ L \\
T &\rightarrow \text{int} \\
L &\rightarrow M_2\ L\ ,\ id \\
L &\rightarrow M_3\ id \\
M_1 &\rightarrow \epsilon \\
M_2 &\rightarrow \epsilon \\
M_3 &\rightarrow \epsilon
\end{aligned}
$$

I_0

$$
\boxed{\begin{aligned}
D' &\rightarrow \bullet\, D \\
D &\rightarrow \bullet\, T\ M_1\ L \\
T &\rightarrow \bullet\, \text{int}
\end{aligned}}
$$

$\big\downarrow\ T$

I_1

$$
\boxed{\begin{aligned}
D &\rightarrow T\ \bullet\ M_1\ L \\
M_1 &\rightarrow \epsilon\ \bullet
\end{aligned}}
$$

The Role of Semantic Analysis
Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

Marker Non-Terminals May Cause Reduce-Reduce Conflicts

Consider the grammar of declaration consisting of non-terminals D (Declaration), T (Type), L (List of identifiers), terminals int, "," and id, and marker non-terminals M_1, M_2, and M_3

$$
\begin{aligned}
D &\rightarrow T\ M_1\ L \\
T &\rightarrow \text{int} \\
L &\rightarrow M_2\ L\ ,\ id \\
L &\rightarrow M_3\ id \\
M_1 &\rightarrow \epsilon \\
M_2 &\rightarrow \epsilon \\
M_3 &\rightarrow \epsilon
\end{aligned}
$$

I_0:

$$
\boxed{\begin{aligned}
D' &\rightarrow \bullet D \\
D &\rightarrow \bullet T\ M_1\ L \\
T &\rightarrow \bullet \text{int}
\end{aligned}}
$$

$I_0 \xrightarrow{T} I_1$

I_1:

$$
\boxed{\begin{aligned}
D &\rightarrow T \bullet M_1\ L \\
M_1 &\rightarrow \epsilon \bullet
\end{aligned}}
$$

$I_1 \xrightarrow{M_1} I_2$

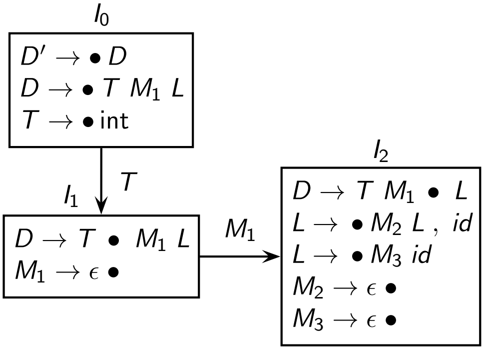
I_2:

$$
\boxed{\begin{aligned}
D &\rightarrow T\ M_1 \bullet L \\
L &\rightarrow \bullet M_2\ L\ ,\ id \\
L &\rightarrow \bullet M_3\ id \\
M_2 &\rightarrow \epsilon \bullet \\
M_3 &\rightarrow \epsilon \bullet
\end{aligned}}
$$

Marker Non-Terminals May Cause Reduce-Reduce Conflicts

Consider the grammar of declaration consisting of non-terminals D (Declaration), T (Type), L (List of identifiers), terminals int, "," and *id*, and marker non-terminals M_1, M_2, and M_3

$$
\begin{aligned}
D &\rightarrow T\ M_1\ L \\
T &\rightarrow \text{int} \\
L &\rightarrow M_2\ L\ ,\ id \\
L &\rightarrow M_3\ id \\
M_1 &\rightarrow \epsilon \\
M_2 &\rightarrow \epsilon \\
M_3 &\rightarrow \epsilon
\end{aligned}
$$



We have reduce-reduce conflict in I_2 because *id* is in the FOLLOW of M_2 and M_3
We can avoid it by rewriting the grammar (see the last slide in this pdf)

SDTS for FOR Loop with BREAK and CONTINUE Statements

$$
\begin{array}{lll}
S_1 & \rightarrow & \textbf{for}\ (E_1;\ E_2;\ E_3) \\
 & & \{\ S_2.increment = getNewLabel() \quad \text{/* needed here because it is inherited */} \\
 & & \quad S_2.loopback = getNewLabel() \quad \text{/* can be moved to the end of the rule */} \\
 & & \quad S_2.exit = getNewLabel() \quad \text{/* needed here because it is inherited */} \\
 & & \} \\
 & & S_2 \\
 & & \{\ t_1 = getNewTemp() \\
 & & \quad c_1 = gen(S_2.loopback, :) \\
 & & \quad c_2 = gen(t_1, =!, E_2.place) \,\|\, gen(\text{if}, t_1, \text{goto}, S_2.exit) \\
 & & \quad c_3 = gen(\text{goto}, S_2.increment) \\
 & & \quad c_4 = gen(S_2.exit, :) \\
 & & \quad S_1.code = E_1.code \,\|\, c_1 \,\|\, E_2.code \,\|\, c_2 \,\|\, S_2.code \,\|\, c_3 \,\|\, E_3.code \,\|\, c_4 \\
 & & \} \\
S & \rightarrow & \textbf{break}\ \{S.code = gen(\text{goto}, S.exit)\} \\
S & \rightarrow & \textbf{continue}\ \{S.code = gen(\text{goto}, S.increment)\}
\end{array}
$$

Type Analysis

Type Analysis

- Type Expressions
- Type Equivalence
- Type Checking and Type Inferencing

The Role of Types

1. Types provide information about
   - the size of data and the interpretation of raw bits, and (the integer value of string of four bytes $\boxed{1111}$ is $4096+256+16+1 = 4369$)
   - the operations allowed on data
2. The type of a variable may be allowed to change during the lifetime of the data
   - Python, AWK allow the same variables to have different types at different program points
   - C/C++ do not allow this; instead they allow implicit *type promotion* and explicit type conversion (aka *type casting*)
3. Types may be known at compile time or only at run time

Most literature conflates (2) and (3) above and use the term *dynamically checked languages* for such languages

Property (2) should be called *flow-sensitive* or *flow-insensitive* types and the terms *static* or *dynamic checking* should be reserved for property (3)

Type System

- A *type system* is a set of rules that assign a unique type to each data item
  - The assigned type may include a *type error*
  - A type system accepts a program if it succeeds in assigning valid non-error types to all data items
- A *sound* type system guarantees that a program accepted by the type system would not have any unchecked type error at run time
  - A sound type system is not required check the types at compile time; the types may well be checked at run time
  - A type system that rejects all programs is vacuously sound

Type Expressions

A type expression describes types of all entities (variables, functions) in a program

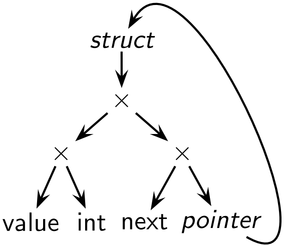
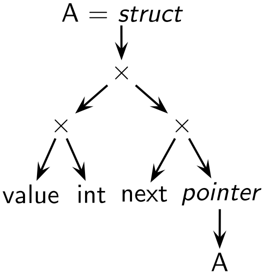
- A basic type such as int, float, void, bool, char is a type expression
- A user defined type name is a type expression
- A type constructor applied to a type expression τ is also a type expression
  These type expressions represent derived types

Type Expressions for Derived Types

- $\textit{array}(k, \tau)$ describes an array of k elements of type τ
  - The size of an array is not a part of the type in C for validation; it is needed for memory allocation
- $\textit{pointer}(\tau)$ describes a pointer to an element of type τ
- $\textit{struct}((f_1, \tau_1), (f_2, \tau_2), \ldots, (f_k, \tau_k))$ describes a structure containing k fields named f_1 to f_k with types τ_1 to τ_k
  - f_1 to f_k must be distinct but τ_1 to τ_k need not be distinct
- $\tau_1 \rightarrow \tau_2$ describes a function that takes arguments described by τ_1 and returns result described by τ_2.
- Given τ_1 and τ_2, $\tau_1 \times \tau_2$ describes the product of the two types
  - Product can be used to represent a list or tuples of type expressions
  - Product is left associative and has a higher precedence than $\rightarrow$

Representing Type Expression

- A type expression can be represented as a graph
- In general, it may contain cycles but we convert it into a tree by naming the target of the back edge and using the name as a node

Declaration	Type Graph	Type Tree
`struct A { int value; struct A *next; } ;`		


```
struct A {
   int value;
   struct A *next;
   } ;
```

- The resulting type expression is written with A as the name of the type expression as $A = \mathit{struct}((\mathrm{value}, \mathrm{int}), (\mathrm{next}, \mathit{pointer}(A)))$

Type Equivalence

- Consider the following declarations

```
struct Person          struct Laptop          struct Car
{                      {                      {
  string name;           string name;           string name;
  float weight;          float weight;          float weight;
};                     };                     };

int A[5][50];          int B[10][20];         int C[100][200];
```

- Are variables of the type `struct Person`, `struct Laptop`, and `struct Car` compatible with each other?

  (i.e., can the value of one be assigned to the other?)

- Are elements of arrays `A`, `B`, and `C` compatible with each other?

  (i.e., can the value of one be assigned to the other?)

Name and Structural Equivalence of Types

- Name Equivalence
  - Same basic types are name equivalent
  - Derived type are name equivalent if they have the same name
    - Every occurrence of a derived type in declarations is given a unique name
- Structural Equivalence
  - Same basic types are structurally equivalent
  - Derived type are structurally equivalent if
    - they are obtained by applying the same type constructors to structurally equivalent types, or
    - one is type name that denotes the other type expressions
- Name equivalence implies structural equivalence but not vice-versa
- C uses structural equivalence for everything except structures

  For structures, it uses name equivalence

Examples of Type Equivalence

- Consider the following declarations

```
struct Person        struct Laptop        struct Car
{                    {                    {
  string name;         string name;         string name;
  float weight;        float weight;        float weight;
} p1, p2;            } l1, l2;            } c1, c2;
```

- Partition of variables
  - under name equivalence: $\{\{\texttt{p1, p2}\}, \{\texttt{l1, l2}\}, \{\texttt{c1, c2}\}\}$
  - under structural equivalence: $\{\{\texttt{p1, p2, l1, l2, c1, c2}\}\}$

SDD for Type Checking

$E \rightarrow \text{char\_const}$	$\{E.type = \text{char}\}$
$E \rightarrow num$	$\{E.type = \text{int}\}$
$E \rightarrow id$	$\{E.type = id.type\}$
$E_1 \rightarrow E_2 \text{ mod } E_3$	$\{$ if $((E_2.type \equiv \text{int})$ && $(E_3.type \equiv \text{int}))$ $E_1.type = \text{int}$ else $E_1.type = \text{type\_error}$ $\}$

$E_1 \rightarrow E_2\ op\ E_3$	$\{$ $E_1.type = \text{type\_error}$
	if $(E_2.type \equiv E_3.type)$
	$\{$ valI $= (E_2.type \equiv \text{int})$; valF $= (E_2.type \equiv \text{float})$
	valB $= (E_2.type \equiv \text{bool})$; opA $= (op.type \equiv arith)$
	opB $= (op.type \equiv \text{bool})$; opR $= (op.type \equiv rel)$
	if (opR && (valI || valF)) $E_1.type = \text{bool}$
	if ((opA && (valI || valF)) || (opB && valB))
	$E_1.type = E_2.type$ $\}$ $\}$

$E_1 \rightarrow E_2[E_3]$	$\{$ if $((E_2.type \equiv array(n,t))$ && $(E_3.type \equiv \text{int}))$ $E_1.type = t$ else $E_1.type = \text{type\_error}$ $\}$
$E_1 \rightarrow *E_2$	$\{$ if $(E_2.type \equiv pointer(t))$ $E_1.type = t$ else $E_1.type = \text{type\_error}$ $\}$

Type Inferencing

- Functional languages do not require separate declarations for variables and types
- Usually, functions are annotated with type information and most other types are inferred from these annotations, the constants, and the operators
- The type expressions in such languages also contain type variables whose values are type expressions
- The values of type variables is inferred by unifying type expressions that are expected to represent the same type

The Role of Semantic Analysis

Examples of Errors

Syntax Directed Definitions

Generating IR

Syntax Directed Translation Schemes

Type Analysis

Name and Scope Analysis

Declaration Processing

Name and Scope Analysis

Scope Analysis: Key Ideas

- Maintain a stack of symbol tables
- At the start of a new scope, push a new symbol table on the stack
  - Beginning of the program ("global" scope)
  - Beginning of every procedure
    The procedure name belongs to the outer scope
  - Beginning of every compound statement
- At the end of every scope, pop the top symbol table from the stack
  (Store it in a persistent data structure)
- For use of a name, look it up in the symbol table starting from the stack top
  - If the name is not found in a symbol table, search in the symbol table below
  - If the same name appears in two symbol tables, the one closer to the top hides the one below

    The symbol table below closer to the top represents the more closely nested procedure and shadows the names in the outer procedures

Access to Non-local Variables

```
int main()
{





















  // body of main
}
```

Nested function in C supported by GCC extension

(Originally supported in Pascal but not supported by C standards)

Access to Non-local Variables

```
int main()
{ void S()
  { int a, x;






    // body of S
  }
  // body of main
}
```

Nested function in C supported by GCC extension

(Originally supported in Pascal but not supported by C standards)

Access to Non-local Variables

```
int main()
{ void S()
  { int a, x;
    void R()
    { int i;



      // body of R }





   // body of S
  }
 // body of main
}
```

Nested function in C supported by GCC extension

(Originally supported in Pascal but not supported by C standards)

Access to Non-local Variables

```
int main()
{ void S()
  { int a, x;
    void R()
    { int i;
      int T()
      { int m,n;
        // body of T
      }
      // body of R }



     // body of S
  }
  // body of main
}
```

Nested function in C supported by GCC extension

(Originally supported in Pascal but not supported by C standards)

Access to Non-local Variables

```
int main()
{ void S()
  { int a, x;
    void R()
    { int i;
      int T()
      { int m,n;
        // body of T
      }
     // body of R }
    void E()
    { // body of E   }




   // body of S
  }
 // body of main
}
```

Nested function in C supported by GCC extension

(Originally supported in Pascal but not supported by C standards)

Access to Non-local Variables

```
int main()
{ void S()
  { int a, x;
    void R()
    { int i;
      int T()
      { int m,n;
        // body of T
      }
      // body of R }
    void E()
    { // body of E   }
    void Q()
    { int a, x;


      // body of Q }
    // body of S
  }
  // body of main
}
```

Nested function in C supported by GCC extension

(Originally supported in Pascal but not supported by C standards)

Access to Non-local Variables

```
int main()
{ void S()
  { int a, x;
    void R()
    { int i;
      int T()
      { int m,n;
        // body of T
      }
      // body of R }
    void E()
    { // body of E   }
    void Q()
    { int a, x;
      int P(int y, int z)
      { int i,j;
        // body of P
      }
      // body of Q }
   // body of S
  }
  // body of main
}
```

Nested function in C supported by GCC extension

(Originally supported in Pascal but not supported by C standards)

Access to Non-local Variables: Static Scope

```
int main()
{ void S()
  { int a, x;
    void R()
    { int i;
      int T()
      { int m,n;
        // body of T
      }
      // body of R }
    void E()
    { // body of E   }
    void Q()
    { int a, x;
      int P(int y, int z)
      { int i,j;
        // body of P
      }
      // body of Q }
    // body of S
  }
  // body of main
}
```

- Under *static scoping* , the names visible at line i in procedure X are:
  - names declared locally within X before line i
  - names declared in procedures enclosing X upto the declaration of X in the program
- A name declared in more closely nested procedure overrides the same name declared in an outer procedure.

Access to Non-local Variables: Static Scope

```
int main()
{ void S()
  { int a, x;
    void R()
    { int i;
      int T()
      { int m,n;
        // body of T
      }
      // body of R }
    void E()
    { // body of E   }
    void Q()
    { int a, x;
      int P(int y, int z)
      { int i,j;
        // body of P
      }
      // body of Q }
   // body of S
  }
 // body of main
}
```

- Under *static scoping* , the names visible at line *i* in procedure *X* are:
  - names declared locally within *X* before line *i*
  - names declared in procedures enclosing *X* upto the declaration of *X* in the program
- A name declared in more closely nested procedure overrides the same name declared in an outer procedure.
- The names visible in the body of `T` are:
  - `T`, `R`, `S`, `main` (enclosing procedure names)
  - `T:m`, `T:n`, `R:i`, `S:a`, and `S:x` (names declared immediately within `T`, `R` and `S`)
    `E` and `Q` are declared within `S` but are *not* visible in `T` (but they are visible in `P`)
  - For call chain `main` $\rightarrow$ `S` $\rightarrow$ `Q` $\rightarrow$ `E` $\rightarrow$ `R` $\rightarrow$ `T`, variables `S:a` and `S:x` are accessed in `T` and not `Q:a` and `Q:x`

Access to Non-local Variables: Dynamic Scope

```
int main()
{ void S()
  { int a, x;
    void R()
    { int i;
      int T()
      { int m,n;
        // body of T
      }
      // body of R }
    void E()
    { // body of E   }
    void Q()
    { int a, x;
      int P(int y, int z)
      { int i,j;
        // body of P
      }
      // body of Q }
   // body of S
  }
 // body of main
}
```

- Under *dynamic scoping* , the names visible at line i in procedure X are:
  - names declared locally within X before line i
  - names declared in procedures enclosing X in a call chain reaching X
- A name declared in more closely nested procedure in the call chain overrides the same name declared in an outer procedure.

Access to Non-local Variables: Dynamic Scope

```
int main()
{ void S()
  { int a, x;
    void R()
    { int i;
      int T()
      { int m,n;
        // body of T
      }
      // body of R }
    void E()
    { // body of E   }
    void Q()
    { int a, x;
      int P(int y, int z)
      { int i,j;
        // body of P
      }
      // body of Q }
    // body of S
  }
  // body of main
}
```

- Under *dynamic scoping* , the names visible at line i in procedure X are:
  - names declared locally within X before line i
  - names declared in procedures enclosing X in a call chain reaching X
- A name declared in more closely nested procedure in the call chain overrides the same name declared in an outer procedure.
- For a call chain `main` $\rightarrow$ `S` $\rightarrow$ `Q` $\rightarrow$ `E` $\rightarrow$ `R` $\rightarrow$ `T` the names visible in the body of `T` are:
  - The names in `T`, `R`, `E`, `Q`, `S` and `main`
  - Variables `S:a` and `S:x` are shadowed by `Q:a` and `Q:x` in `T`

Scope Analysis Demo for Static Scope

- scope-analysis.y

Scope Analysis: Grammar

$$
\begin{aligned}
\text{Program} &\rightarrow \text{DL SL} \\
\text{DL} &\rightarrow \text{DL D} \mid \epsilon \\
\text{D} &\rightarrow \text{T } \textit{id} \\
\text{D} &\rightarrow \text{T } \textit{id} \text{ (PL) \{ DL SL \}} \\
\text{T} &\rightarrow \text{int} \mid \text{void} \\
\text{PL} &\rightarrow \text{PL , P} \mid \text{P} \\
\text{P} &\rightarrow \text{T } \textit{id} \\
\text{SL} &\rightarrow \text{SL Call} \mid \epsilon \\
\text{Call} &\rightarrow \textit{id} \text{ (AL) ;} \\
\text{AL} &\rightarrow \text{AL , } \textit{id} \mid \textit{id}
\end{aligned}
$$

We consider a simplified grammar in which

- DL denotes a list of declarations
- D denotes a declaration

  For simplicity, we assume that a single name can be declared in a declaration
- T denotes a type declaration
- PL denotes a list of formal parameters
- P denotes a formal parameter
- SL denotes a list of statement

  For simplicity, we consider only a call statement
- AL denotes a list of actual parameters

Scope Analysis: SDTS

$$\begin{array}{rll}
\text{Program} \rightarrow & & \text{DL SL} \\
\text{DL} \rightarrow & \text{DL D} \mid \epsilon & \\
\text{D} \rightarrow & \text{T } \mathit{id} & \\
\text{D} \rightarrow & \text{T } \mathit{id} & \\
 & & (\text{ PL }) \; \{ \text{ DL SL } \} \\
\text{T} \rightarrow & \text{int} & \mid \text{void} \\
\text{PL} \rightarrow & \text{PL , P} \mid \text{P} & \\
\text{P} \rightarrow & \text{T } \mathit{id} & \\
\text{SL} \rightarrow & \text{SL Call} \mid \epsilon & \\
\text{Call} \rightarrow & \mathit{id} & (\text{ AL }) \; ; \\
\text{AL} \rightarrow & \text{AL , } \mathit{id} & \mid \mathit{id}
\end{array}$$

Scope Analysis: SDTS

Program → { push_new_symtab(); } DL SL

DL → DL D | ϵ

D → T *id*

D → T *id* (PL) { DL SL }

T → int | void

PL → PL , P | P

P → T *id*

SL → SL Call | ϵ

Call → *id* (AL) ;

AL → AL , *id* | *id*

Scope Analysis: SDTS

$$\text{Program} \rightarrow \{\ \text{push\_new\_symtab}();\ \}\ \text{DL SL}$$

$$\text{DL} \rightarrow \text{DL D} \mid \epsilon$$

$$\text{D} \rightarrow \text{T}\ \textit{id}\ \{\ \text{add\_var\_to\_symtab}(\textit{id.name}, \text{T}.\textit{name})\}$$

$$\text{D} \rightarrow \text{T}\ \textit{id} \qquad\qquad (\ \text{PL}\ )\ \{\ \text{DL SL}\ \}$$

$$\text{T} \rightarrow \text{int}\ \{\text{T}.\textit{name} = \text{int};\ \}\ \mid \text{void}\ \{\text{T}.\textit{name} = \text{void};\ \}$$

$$\text{PL} \rightarrow \text{PL}\ ,\ \text{P} \mid \text{P}$$

$$\text{P} \rightarrow \text{T}\ \textit{id}$$

$$\text{SL} \rightarrow \text{SL Call} \mid \epsilon$$

$$\text{Call} \rightarrow \textit{id} \qquad\qquad (\ \text{AL}\ )\ ;$$

$$\text{AL} \rightarrow \text{AL}\ ,\ \textit{id} \qquad\qquad \mid \textit{id}$$

Scope Analysis: SDTS

$$
\begin{aligned}
\text{Program} &\rightarrow \{\ \text{push\_new\_symtab}();\ \}\ \text{DL SL} \\
\text{DL} &\rightarrow \text{DL D} \mid \epsilon \\
\text{D} &\rightarrow \text{T}\ \textit{id}\ \{\ \text{add\_var\_to\_symtab}(\textit{id}.\textit{name}, \text{T}.\textit{name})\} \\
\text{D} &\rightarrow \text{T}\ \textit{id}\ \{\ \text{add\_proc\_to\_symtab}(\textit{id}.\textit{name}, \text{T}.\textit{name});\ \} \\
& \qquad\qquad (\ \text{PL}\ )\ \{\ \text{DL SL}\ \} \\
\\
\text{T} &\rightarrow \text{int}\ \{\text{T}.\textit{name} = \text{int};\ \}\ \mid \text{void}\ \{\text{T}.\textit{name} = \text{void};\ \} \\
\text{PL} &\rightarrow \text{PL}\ ,\ \text{P} \mid \text{P} \\
\text{P} &\rightarrow \text{T}\ \textit{id} \\
\text{SL} &\rightarrow \text{SL Call} \mid \epsilon \\
\text{Call} &\rightarrow \textit{id}\ (\ \text{AL}\ )\ ; \\
\text{AL} &\rightarrow \text{AL}\ ,\ \textit{id} \mid \textit{id}
\end{aligned}
$$

Scope Analysis: SDTS

$$
\begin{aligned}
\text{Program} &\rightarrow \{\ \text{push\_new\_symtab}();\ \}\ \text{DL SL} \\
\text{DL} &\rightarrow \text{DL D} \mid \epsilon \\
\text{D} &\rightarrow \text{T}\ \mathit{id}\ \{\ \text{add\_var\_to\_symtab}(\mathit{id}.\mathit{name}, \text{T}.\mathit{name})\} \\
\text{D} &\rightarrow \text{T}\ \mathit{id}\ \{\ \text{add\_proc\_to\_symtab}(\mathit{id}.\mathit{name}, \text{T}.\mathit{name});\ \} \\
&\quad\ \{\ \text{push\_new\_symtab}();\ \}\ (\ \text{PL}\ )\ \{\ \text{DL SL}\ \} \\
\\
\text{T} &\rightarrow \text{int}\ \{\text{T}.\mathit{name} = \text{int};\ \}\ \mid \text{void}\ \{\text{T}.\mathit{name} = \text{void};\ \} \\
\text{PL} &\rightarrow \text{PL}\ ,\ \text{P} \mid \text{P} \\
\text{P} &\rightarrow \text{T}\ \mathit{id} \\
\text{SL} &\rightarrow \text{SL Call} \mid \epsilon \\
\text{Call} &\rightarrow \mathit{id}\ (\ \text{AL}\ )\ ; \\
\text{AL} &\rightarrow \text{AL}\ ,\ \mathit{id} \mid \mathit{id}
\end{aligned}
$$

Scope Analysis: SDTS

$$
\begin{array}{rl}
\text{Program} \rightarrow & \{\ \text{push\_new\_symtab}();\ \}\ \text{DL SL} \\
\text{DL} \rightarrow & \text{DL D} \mid \epsilon \\
\text{D} \rightarrow & \text{T}\ \mathit{id}\ \{\ \text{add\_var\_to\_symtab}(\mathit{id.name}, \text{T}.\mathit{name})\} \\
\text{D} \rightarrow & \text{T}\ \mathit{id}\ \{\ \text{add\_proc\_to\_symtab}(\mathit{id.name}, \text{T}.\mathit{name});\ \} \\
& \{\ \text{push\_new\_symtab}();\ \}\ (\ \text{PL}\ )\ \{\ \text{DL SL}\ \} \\
& \{\ \text{pop\_symtab}();\ \} \\
\text{T} \rightarrow & \text{int}\ \{\text{T}.\mathit{name} = \text{int};\ \}\ \mid \text{void}\ \{\text{T}.\mathit{name} = \text{void};\ \} \\
\text{PL} \rightarrow & \text{PL}\ ,\ \text{P} \mid \text{P} \\
\text{P} \rightarrow & \text{T}\ \mathit{id} \\
\text{SL} \rightarrow & \text{SL Call} \mid \epsilon \\
\text{Call} \rightarrow & \mathit{id}\ (\ \text{AL}\ )\ ; \\
\text{AL} \rightarrow & \text{AL}\ ,\ \mathit{id} \mid \mathit{id}
\end{array}
$$

Pop and move it to a persistent storage for later phases

IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Semantic Analysis

Section:

The Role of Semantic Analysis

Examples of Errors

Syntax Directed Definitions

Generating IR

Syntax Directed Translation Schemes

Type Analysis

Name and Scope Analysis

Declaration Processing

Scope Analysis: SDTS

$$
\begin{array}{rl}
\text{Program} \rightarrow & \{\ \text{push\_new\_symtab}();\ \}\ \text{DL SL} \\
\text{DL} \rightarrow & \text{DL D} \mid \epsilon \\
\text{D} \rightarrow & \text{T}\ id\ \{\ \text{add\_var\_to\_symtab}(id.name, \text{T}.name)\} \\
\text{D} \rightarrow & \text{T}\ id\ \{\ \text{add\_proc\_to\_symtab}(id.name, \text{T}.name);\ \} \\
& \{\ \text{push\_new\_symtab}();\ \}\ (\ \text{PL}\ )\ \{\ \text{DL SL}\ \} \\
& \{\ \text{pop\_symtab}();\ \} \\
\text{T} \rightarrow & \text{int}\ \{\text{T}.name = \text{int};\ \}\ \mid \text{void}\ \{\text{T}.name = \text{void};\ \} \\
\text{PL} \rightarrow & \text{PL}\ ,\ \text{P} \mid \text{P} \\
\text{P} \rightarrow & \text{T}\ id\ \{\ \text{add\_param\_to\_symtab}(id.name, \text{T}.name);\ \} \\
\text{SL} \rightarrow & \text{SL Call} \mid \epsilon \\
\text{Call} \rightarrow & id\ (\ \text{AL}\ )\ ; \\
\text{AL} \rightarrow & \text{AL}\ ,\ id \mid id
\end{array}
$$

Scope Analysis: SDTS

$$
\begin{array}{rcl}
\text{Program} & \rightarrow & \{\ \text{push\_new\_symtab}();\ \}\ \text{DL SL} \\
\text{DL} & \rightarrow & \text{DL D} \mid \epsilon \\
\text{D} & \rightarrow & \text{T}\ \textit{id}\ \{\ \text{add\_var\_to\_symtab}(\textit{id.name}, \text{T}.\textit{name})\} \\
\text{D} & \rightarrow & \text{T}\ \textit{id}\ \{\ \text{add\_proc\_to\_symtab}(\textit{id.name}, \text{T}.\textit{name});\ \} \\
 & & \{\ \text{push\_new\_symtab}();\ \}\ (\ \text{PL}\ )\ \{\ \text{DL SL}\ \} \\
 & & \{\ \text{pop\_symtab}();\ \} \\
\text{T} & \rightarrow & \text{int}\ \{\text{T}.\textit{name} = \text{int};\ \} \mid \text{void}\ \{\text{T}.\textit{name} = \text{void};\ \} \\
\text{PL} & \rightarrow & \text{PL}\ ,\ \text{P} \mid \text{P} \\
\text{P} & \rightarrow & \text{T}\ \textit{id}\ \{\ \text{add\_param\_to\_symtab}(\textit{id.name}, \text{T}.\textit{name});\ \} \\
\text{SL} & \rightarrow & \text{SL Call} \mid \epsilon \\
\text{Call} & \rightarrow & \textit{id}\ \{\ \text{lookup}(\textit{id.name});\ \}\ (\ \text{AL}\ )\ ; \\
\text{AL} & \rightarrow & \text{AL}\ ,\ \textit{id} \mid \textit{id}
\end{array}
$$

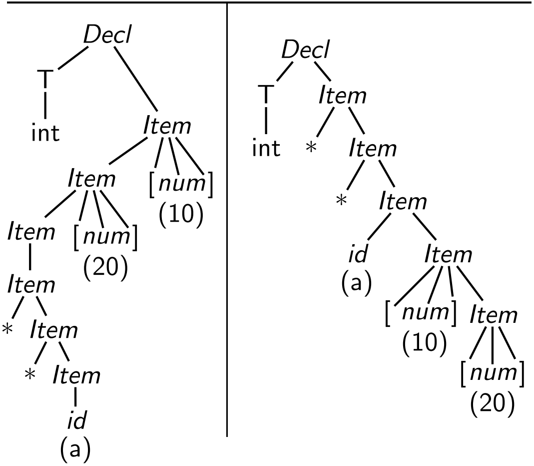
Scope Analysis: SDTS

$$
\begin{aligned}
\text{Program} &\rightarrow \{\ \text{push\_new\_symtab}();\ \}\ \text{DL SL} \\
\text{DL} &\rightarrow \text{DL D} \mid \epsilon \\
\text{D} &\rightarrow \text{T}\ id\ \{\ \text{add\_var\_to\_symtab}(id.name, \text{T}.name)\} \\
\text{D} &\rightarrow \text{T}\ id\ \{\ \text{add\_proc\_to\_symtab}(id.name, \text{T}.name);\ \} \\
&\quad \{\ \text{push\_new\_symtab}();\ \}\ (\ \text{PL}\ )\ \{\ \text{DL SL}\ \} \\
&\quad \{\ \text{pop\_symtab}();\ \} \\
\text{T} &\rightarrow \text{int}\ \{\text{T}.name = \text{int};\ \} \mid \text{void}\ \{\text{T}.name = \text{void};\ \} \\
\text{PL} &\rightarrow \text{PL}\ ,\ \text{P} \mid \text{P} \\
\text{P} &\rightarrow \text{T}\ id\ \{\ \text{add\_param\_to\_symtab}(id.name, \text{T}.name);\ \} \\
\text{SL} &\rightarrow \text{SL Call} \mid \epsilon \\
\text{Call} &\rightarrow id\ \{\ \text{lookup}(id.name);\ \}\ (\ \text{AL}\ )\ ; \\
\text{AL} &\rightarrow \text{AL}\ ,\ id\ \{\ \text{lookup}(id.name);\ \} \mid id\ \{\ \text{lookup}(id.name);\ \}
\end{aligned}
$$

Declaration Processing

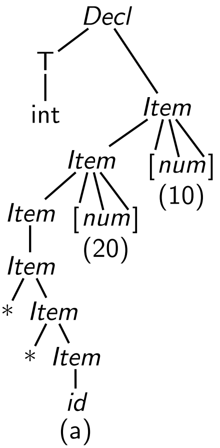
Processing C Declarations

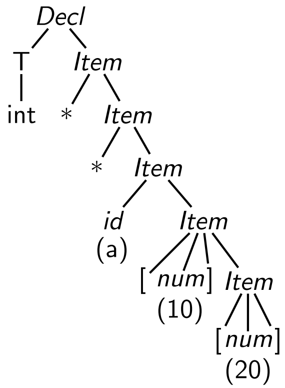
Example Declaration: int **a[20][10];
Two of the many possible parse trees



Processing C Declarations

Example Declaration: int $**$a[20][10];
Two of the many possible parse trees





Difficulties in implementing a syntax directed translation scheme

- Type constructor '$*$' appears before *id* whereas *array* appears after *id*
- Both constructors may appear together for the same *id*
- Final type can be entered in the symbol table only on seeing *id* but the type expression is not complete when *id* is seen
- A combination of synthesized and inherited attributes is needed

Processing C Declarations

- Basic types
- Derived types using type constructors
  (such as arrays, structs, pointer dereferences, address expressions)
- Representing types using type expressions (drawn as trees)

Processing C Declarations

- Basic types
- Derived types using type constructors
  (such as arrays, structs, pointer dereferences, address expressions)
- Representing types using type expressions (drawn as trees)

  int $**$a[20][10];

  Row major representation of arrays in C

  - 20 rows of (i.e. 20 arrays) where
  - each row is an array of 10 double pointers to int
  - the tree is right-recursive for type constructor *array*

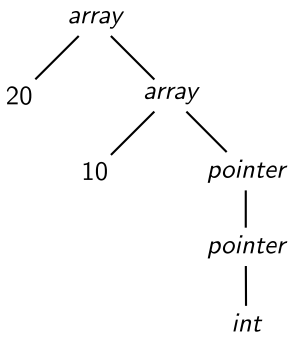
Processing C Declarations

- Basic types
- Derived types using type constructors
  (such as arrays, structs, pointer dereferences, address expressions)
- Representing types using type expressions (drawn as trees)

  `int **a[20][10];`

  Row major representation of arrays in C

  - 20 rows of (i.e. 20 arrays) where
  - each row is an array of 10 double pointers to int
  - the tree is right-recursive for type constructor *array*



Processing C Array Declarations

```
int a[20][10];
```

$$Decl \rightarrow \mathsf{T}\ Item\ \ ;$$

$$\mathsf{T} \rightarrow \mathsf{int}\ \big|\ double$$

$$Item \rightarrow id\ \big|\ Item\ [\ num\ ]$$

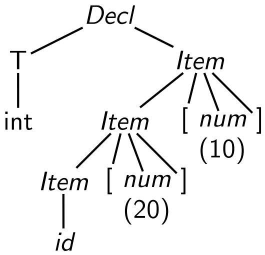
Processing C Array Declarations

```
int a[20][10];
```

$$Decl \rightarrow \mathsf{T}\ Item\ \ ;$$

$$\mathsf{T} \rightarrow \mathsf{int}\ \big|\ double$$

$$Item \rightarrow id\ \big|\ Item\ [\ num\ ]$$



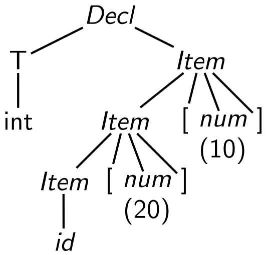
Processing C Array Declarations

```
int a[20][10];
```

$$Decl \rightarrow \text{T}\ Item\ \ ;$$

$$\text{T} \rightarrow \text{int} \mid double$$

$$Item \rightarrow id \mid Item\ [\ num\ ]$$



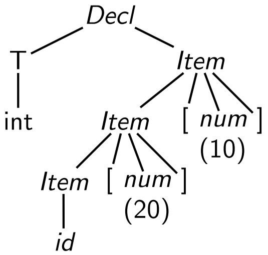
Inconvenient layout for 20 arrays of arrays of 10 ints

Dimensions are collected by a left-recursive rule

Processing C Array Declarations

```
int a[20][10];
```

$$
\begin{aligned}
\mathit{Decl} &\rightarrow \mathrm{T}\ \mathit{Item}\ \ ;\\
\mathrm{T} &\rightarrow \mathrm{int}\ \big|\ \mathit{double}\\
\mathit{Item} &\rightarrow \mathit{id}\ \big|\ \mathit{Item}\ [\ \mathit{num}\ ]
\end{aligned}
$$



Inconvenient layout for 20 arrays of arrays of 10 ints

Dimensions are collected by a left-recursive rule

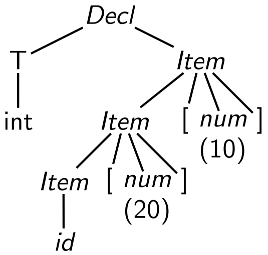
```
int a[20][10];
```

$$
\begin{aligned}
\mathit{Decl} &\rightarrow \mathrm{T}\ \mathit{Item}\ \ ;\\
\mathrm{T} &\rightarrow \mathrm{int}\ \big|\ \mathit{double}\\
\mathit{Item} &\rightarrow \mathit{id}\ \big|\ \mathit{id}\ \mathit{ListDim}\\
\mathit{ListDim} &\rightarrow [\ \mathit{num}\ ]\ \big|\ [\ \mathit{num}\ ]\ \mathit{ListDim}
\end{aligned}
$$

Processing C Array Declarations

```
int a[20][10];
```

$$\mathit{Decl} \rightarrow \mathrm{T}\ \mathit{Item}\ \ ;$$
$$\mathrm{T} \rightarrow \mathrm{int} \mid \mathit{double}$$
$$\mathit{Item} \rightarrow \mathit{id} \mid \mathit{Item}\ [\ \mathit{num}\ ]$$

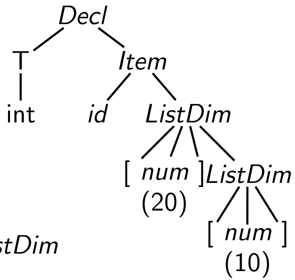


Inconvenient layout for 20 arrays of arrays of 10 ints

Dimensions are collected by a left-recursive rule

```
int a[20][10];
```

$$\mathit{Decl} \rightarrow \mathrm{T}\ \mathit{Item}\ \ ;$$
$$\mathrm{T} \rightarrow \mathrm{int} \mid \mathit{double}$$
$$\mathit{Item} \rightarrow \mathit{id} \mid \mathit{id}\ \mathit{ListDim}$$
$$\mathit{ListDim} \rightarrow [\ \mathit{num}\ ] \mid [\ \mathit{num}\ ]\ \mathit{ListDim}$$



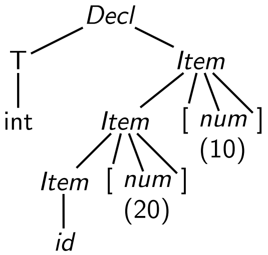
Processing C Array Declarations

```
int a[20][10];
```

$$Decl \rightarrow \mathrm{T}\ Item\ \ ;$$

$$\mathrm{T} \rightarrow \mathrm{int}\ \big|\ double$$

$$Item \rightarrow id\ \big|\ Item\ [\ num\ ]$$



Inconvenient layout for 20 arrays of arrays of 10 ints

Dimensions are collected by a left-recursive rule

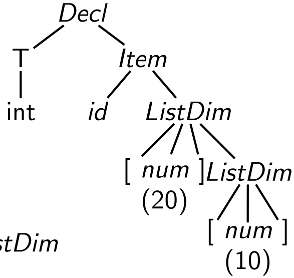
```
int a[20][10];
```

$$Decl \rightarrow \mathrm{T}\ Item\ \ ;$$

$$\mathrm{T} \rightarrow \mathrm{int}\ \big|\ double$$

$$Item \rightarrow id\ \big|\ id\ ListDim$$

$$ListDim \rightarrow [\ num\ ]\ \big|\ [\ num\ ]\ ListDim$$



Convenient layout for 20 arrays of arrays of 10 ints

Dimensions are collected by a right-recursive rule

SDTS for Processing C Array Declarations: Identifying Type

Attribute	Description	Type
$X.bt$	Base type	inherited for $X = I$ and $X = L$, synthesized for $X = T$
$X.dt$	Derived type	synthesized
$X.v$	Value	synthesized
$X.nm$	Name	synthesized

$$D \rightarrow T \quad I\,;$$

$$T \rightarrow \text{int}$$

$$T \rightarrow double$$

$$I \rightarrow id$$

$$I \rightarrow id \quad L$$

$$L \rightarrow [\ num\ ]$$

$$L_1 \rightarrow [\ num\ ] \quad L_2$$

IIT Bombay
cs302: Implementation of Programming Languages

Topic:
Semantic Analysis

Section:
The Role of Semantic Analysis
Examples of Errors
Syntax Directed Definitions
Generating IR
Syntax Directed Translation Schemes
Type Analysis
Name and Scope Analysis
Declaration Processing

SDTS for Processing C Array Declarations: Identifying Type

Attribute	Description	Type
$X.bt$	Base type	inherited for $X = I$ and $X = L$, synthesized for $X = T$
$X.dt$	Derived type	synthesized
$X.v$	Value	synthesized
$X.nm$	Name	synthesized

$$D \rightarrow T \ \{I.bt = T.bt\} \ I \ ;$$

$$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$$

$$T \rightarrow double \ \{T.bt = double\}$$

$$I \rightarrow id$$

$$I \rightarrow id \qquad L$$

$$L \rightarrow [\ num\ ]$$

$$L_1 \rightarrow [\ num\ ] \qquad L_2$$

SDTS for Processing C Array Declarations: Identifying Type

Attribute	Description	Type
$X.bt$	Base type	inherited for $X = I$ and $X = L$, synthesized for $X = T$
$X.dt$	Derived type	synthesized
$X.v$	Value	synthesized
$X.nm$	Name	synthesized

$$D \rightarrow T \; \{I.bt = T.bt\} \; I \, ; \; \{Enter\_In\_Symtab(I.nm, I.dt)\}$$

$$T \rightarrow \text{int} \; \{T.bt = \text{int}\}$$

$$T \rightarrow double \; \{T.bt = double\}$$

$$I \rightarrow id$$

$$I \rightarrow id \qquad L$$

$$L \rightarrow [\, num \,]$$

$$L_1 \rightarrow [\, num \,] \qquad L_2$$

SDTS for Processing C Array Declarations: Identifying Type

Attribute	Description	Type
$X.bt$	Base type	inherited for $X = I$ and $X = L$, synthesized for $X = T$
$X.dt$	Derived type	synthesized
$X.v$	Value	synthesized
$X.nm$	Name	synthesized

$$D \rightarrow T \;\{I.bt = T.bt\}\; I\;;\; \{Enter\_In\_Symtab(I.nm, I.dt)\}$$

$$T \rightarrow \text{int}\; \{T.bt = \text{int}\}$$

$$T \rightarrow double\; \{T.bt = double\}$$

$$I \rightarrow id\; \{I.dt = I.bt;\; I.nm = id.nm\}$$

$$I \rightarrow id \qquad\qquad L$$

$$L \rightarrow [\; num\;]$$

$$L_1 \rightarrow [\; num\;] \qquad\qquad L_2$$

SDTS for Processing C Array Declarations: Identifying Type

Attribute	Description	Type
$X.bt$	Base type	inherited for $X = I$ and $X = L$, synthesized for $X = T$
$X.dt$	Derived type	synthesized
$X.v$	Value	synthesized
$X.nm$	Name	synthesized

$$D \rightarrow T \ \{I.bt = T.bt\} \ \ I\, ; \ \{Enter\_In\_Symtab(I.nm, I.dt)\}$$

$$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$$

$$T \rightarrow double \ \{T.bt = double\}$$

$$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$$

$$I \rightarrow id \ \{L.bt = I.bt\} \ \ L$$

$$L \rightarrow [\, num \,]$$

$$L_1 \rightarrow [\, num \,] \qquad\qquad L_2$$

SDTS for Processing C Array Declarations: Identifying Type

Attribute	Description	Type
$X.bt$	Base type	inherited for $X = I$ and $X = L$, synthesized for $X = T$
$X.dt$	Derived type	synthesized
$X.v$	Value	synthesized
$X.nm$	Name	synthesized

$$D \rightarrow T \ \{I.bt = T.bt\} \ \ I \ ; \ \{Enter\_In\_Symtab(I.nm, I.dt)\}$$

$$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$$

$$T \rightarrow double \ \{T.bt = double\}$$

$$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$$

$$I \rightarrow id \ \{L.bt = I.bt\} \ \ L \ \ \{I.dt = L.dt; \ I.nm = id.nm\}$$

$$L \rightarrow [\ num\ ]$$

$$L_1 \rightarrow [\ num\ ] \qquad\qquad\qquad L_2$$

SDTS for Processing C Array Declarations: Identifying Type

Attribute	Description	Type
$X.bt$	Base type	inherited for $X = I$ and $X = L$, synthesized for $X = T$
$X.dt$	Derived type	synthesized
$X.v$	Value	synthesized
$X.nm$	Name	synthesized

$$D \rightarrow T \ \{I.bt = T.bt\} \ I \ ; \ \{Enter\_In\_Symtab(I.nm, I.dt)\}$$

$$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$$

$$T \rightarrow double \ \{T.bt = double\}$$

$$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$$

$$I \rightarrow id \ \{L.bt = I.bt\} \ L \ \{I.dt = L.dt; \ I.nm = id.nm\}$$

$$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$$

$$L_1 \rightarrow [\ num\ ] \qquad L_2$$

SDTS for Processing C Array Declarations: Identifying Type

Attribute	Description	Type
$X.bt$	Base type	inherited for $X = I$ and $X = L$, synthesized for $X = T$
$X.dt$	Derived type	synthesized
$X.v$	Value	synthesized
$X.nm$	Name	synthesized

$$D \rightarrow T \ \{I.bt = T.bt\} \ I \ ; \ \{Enter\_In\_Symtab(I.nm, I.dt)\}$$

$$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$$

$$T \rightarrow double \ \{T.bt = double\}$$

$$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$$

$$I \rightarrow id \ \{L.bt = I.bt\} \ L \ \{I.dt = L.dt; \ I.nm = id.nm\}$$

$$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$$

$$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\} \ L_2$$

SDTS for Processing C Array Declarations: Identifying Type

Attribute	Description	Type
$X.bt$	Base type	inherited for $X = I$ and $X = L$, synthesized for $X = T$
$X.dt$	Derived type	synthesized
$X.v$	Value	synthesized
$X.nm$	Name	synthesized

$$D \rightarrow T \ \{I.bt = T.bt\} \ I\,; \ \{Enter\_In\_Symtab(I.nm, I.dt)\}$$

$$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$$

$$T \rightarrow double \ \{T.bt = double\}$$

$$I \rightarrow id \ \{I.dt = I.bt;\ I.nm = id.nm\}$$

$$I \rightarrow id \ \{L.bt = I.bt\} \ L \ \{I.dt = L.dt;\ I.nm = id.nm\}$$

$$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$$

$$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\} \ L_2 \ \{L_1.dt = array(num.v, L_2.dt)\}$$

SDTS for Processing C Array Declarations: Identifying Type

$\boxed{\texttt{int a[20][10];}}$

$$D \rightarrow T \ \{I.bt = T.bt\} \ I \ ;$$
$$\{Enter\_In\_Symtab(I.nm, I.dt)\}$$
$$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$$
$$T \rightarrow double \ \{T.bt = double\}$$
$$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$$
$$I \rightarrow id \ \{L.bt = I.bt\} \ L$$
$$\{I.dt = L.dt; \ I.nm = id.nm)\}$$
$$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$$
$$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\}$$
$$L_2 \ \{L_1.dt = array(num.v, L_2.dt)\}$$

SDTS for Processing C Array Declarations: Identifying Type

```
int a[20][10];
```

$D \rightarrow T \ \{I.bt = T.bt\} \ I \, ;$

$\quad \{Enter\_In\_Symtab(I.nm, I.dt)\}$

$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$

$T \rightarrow double \ \{T.bt = double\}$

$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$

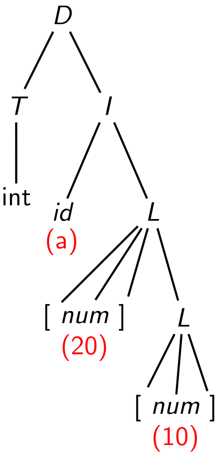
$I \rightarrow id \ \{L.bt = I.bt\} \ L$

$\quad \{I.dt = L.dt; \ I.nm = id.nm)\}$

$L \rightarrow [\, num \,] \ \{L.dt = array(num.v, L.bt)\}$

$L_1 \rightarrow [\, num \,] \ \{L_2.bt = L_1.bt\}$

$\quad L_2 \ \{L_1.dt = array(num.v, L_2.dt)\}$



SDTS for Processing C Array Declarations: Identifying Type

```
int a[20][10];
```

$D \rightarrow T\ \{I.bt = T.bt\}\ I\ ;$

$\quad \{Enter\_In\_Symtab(I.nm, I.dt)\}$

$T \rightarrow \text{int}\ \{T.bt = \text{int}\}$

$T \rightarrow double\ \{T.bt = double\}$

$I \rightarrow id\ \{I.dt = I.bt;\ I.nm = id.nm\}$

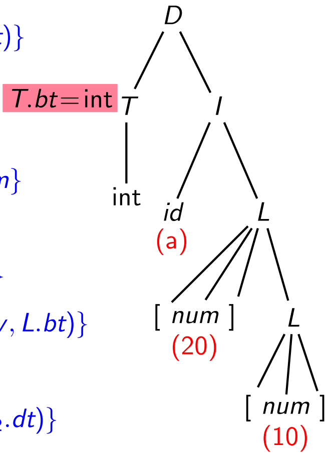
$I \rightarrow id\ \{L.bt = I.bt\}\ L$

$\quad \{I.dt = L.dt;\ I.nm = id.nm)\}$

$L \rightarrow [\ num\ ]\ \{L.dt = array(num.v, L.bt)\}$

$L_1 \rightarrow [\ num\ ]\ \{L_2.bt = L_1.bt\}$

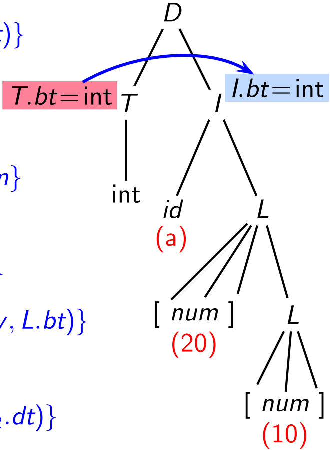
$\quad L_2\ \{L_1.dt = array(num.v, L_2.dt)\}$



SDTS for Processing C Array Declarations: Identifying Type

```
int a[20][10];
```

$$D \rightarrow T \ \{I.bt = T.bt\} \ I\ ;$$
$$\{Enter\_In\_Symtab(I.nm, I.dt)\}$$
$$T \rightarrow \mathrm{int} \ \{T.bt = \mathrm{int}\}$$
$$T \rightarrow double \ \{T.bt = double\}$$
$$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$$
$$I \rightarrow id \ \{L.bt = I.bt\} \ L$$
$$\{I.dt = L.dt; \ I.nm = id.nm)\}$$
$$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$$
$$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\}$$
$$L_2 \ \{L_1.dt = array(num.v, L_2.dt)\}$$



SDTS for Processing C Array Declarations: Identifying Type

```
int a[20][10];
```

$D \rightarrow T \ \{I.bt = T.bt\} \ I \,;$

$\quad \{Enter\_In\_Symtab(I.nm, I.dt)\}$

$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$

$T \rightarrow double \ \{T.bt = double\}$

$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$

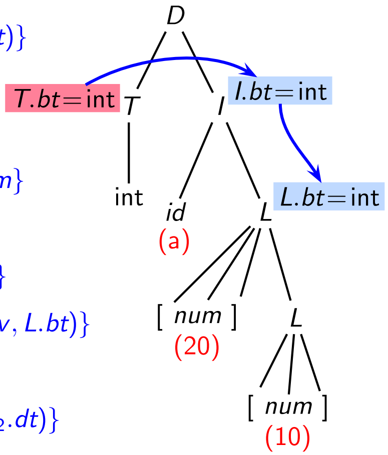
$I \rightarrow id \ \{L.bt = I.bt\} \ L$

$\quad \{I.dt = L.dt; \ I.nm = id.nm)\}$

$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$

$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\}$

$\quad L_2 \ \{L_1.dt = array(num.v, L_2.dt)\}$



SDTS for Processing C Array Declarations: Identifying Type

```
int a[20][10];
```

$D \rightarrow T \ \{I.bt = T.bt\} \ I \ ;$

$\quad \{Enter\_In\_Symtab(I.nm, I.dt)\}$

$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$

$T \rightarrow double \ \{T.bt = double\}$

$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$

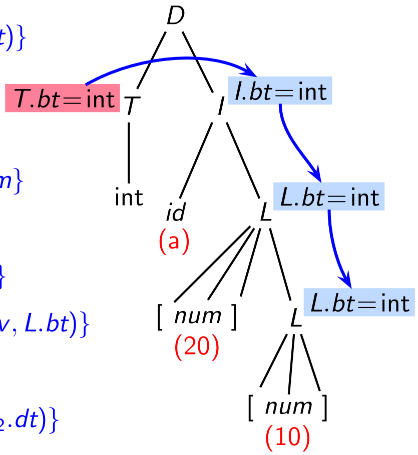
$I \rightarrow id \ \{L.bt = I.bt\} \ L$

$\quad \{I.dt = L.dt; \ I.nm = id.nm)\}$

$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$

$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\}$

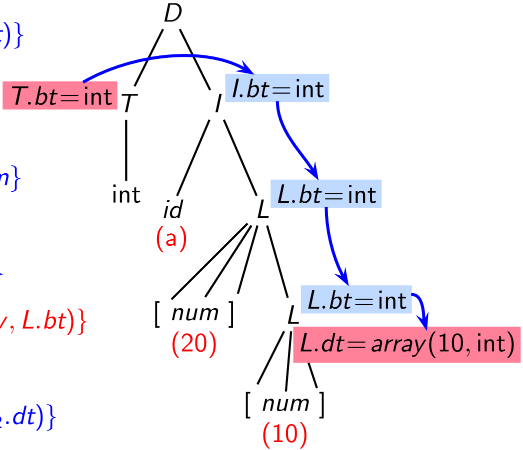
$\quad L_2 \ \{L_1.dt = array(num.v, L_2.dt)\}$



SDTS for Processing C Array Declarations: Identifying Type

```
int a[20][10];
```

$$D \rightarrow T \ \{I.bt = T.bt\} \ I \ ;$$
$$\{Enter\_In\_Symtab(I.nm, I.dt)\}$$
$$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$$
$$T \rightarrow double \ \{T.bt = double\}$$
$$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$$
$$I \rightarrow id \ \{L.bt = I.bt\} \ L$$
$$\{I.dt = L.dt; \ I.nm = id.nm)\}$$
$$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$$
$$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\}$$
$$L_2 \ \{L_1.dt = array(num.v, L_2.dt)\}$$



SDTS for Processing C Array Declarations: Identifying Type

```
int a[20][10];
```

$D \rightarrow T \ \{I.bt = T.bt\} \ I \ ;$

$\qquad \{Enter\_In\_Symtab(I.nm, I.dt)\}$

$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$

$T \rightarrow double \ \{T.bt = double\}$

$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$

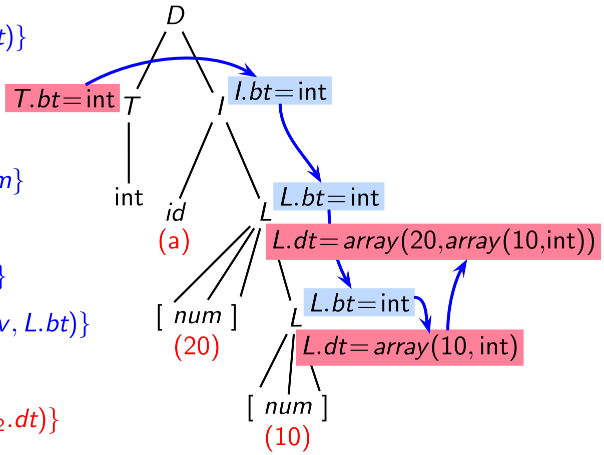
$I \rightarrow id \ \{L.bt = I.bt\} \ L$

$\qquad \{I.dt = L.dt; \ I.nm = id.nm)\}$

$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$

$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\}$

$\qquad L_2 \ \{L_1.dt = array(num.v, L_2.dt)\}$



SDTS for Processing C Array Declarations: Identifying Type

```
int a[20][10];
```

$D \rightarrow T \ \{I.bt = T.bt\} \ I \ ;$

$\quad \{Enter\_In\_Symtab(I.nm, I.dt)\}$

$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$

$T \rightarrow double \ \{T.bt = double\}$

$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$

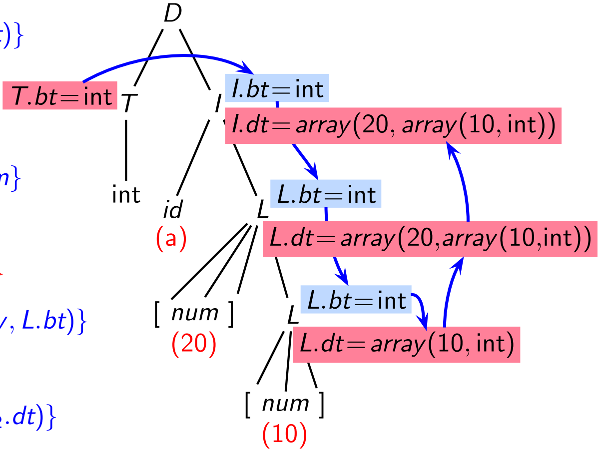
$I \rightarrow id \ \{L.bt = I.bt\} \ L$

$\quad \{I.dt = L.dt; \ I.nm = id.nm)\}$

$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$

$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\}$

$\quad L_2 \ \{L_1.dt = array(num.v, L_2.dt)\}$



SDTS for Processing C Array Declarations: Identifying Type

```
int a[20][10];
```

$D \rightarrow T \ \{I.bt = T.bt\} \ I \ ;$

$\qquad \{Enter\_In\_Symtab(I.nm, I.dt)\}$

$T \rightarrow \text{int} \ \{T.bt = \text{int}\}$

$T \rightarrow double \ \{T.bt = double\}$

$I \rightarrow id \ \{I.dt = I.bt; \ I.nm = id.nm\}$

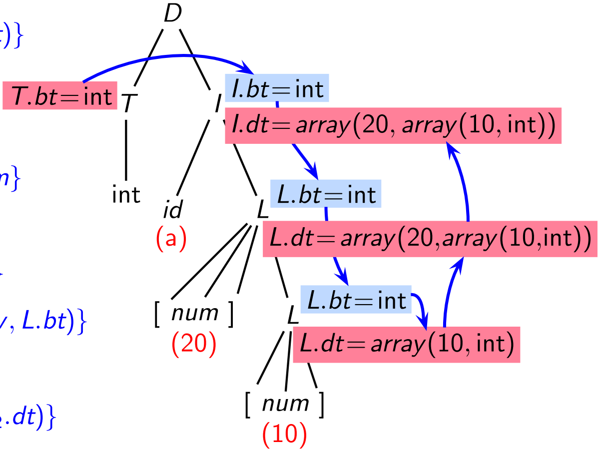
$I \rightarrow id \ \{L.bt = I.bt\} \ L$

$\qquad \{I.dt = L.dt; \ I.nm = id.nm)\}$

$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt)\}$

$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\}$

$\qquad L_2 \ \{L_1.dt = array(num.v, L_2.dt)\}$



C Array Size Calculations

Attribute	Description	Type
$X.bt$	Base type	inherited for $X = I$ and $X = L$, synthesized for $X = T$
$X.dt$	Derived type	synthesized
$X.v$	Value	synthesized
$X.s$	Size	synthesized
$X.nm$	Name	synthesized
$X.w$	Width	inherited for $X = I$, synthesized for $X = T$

$$D \rightarrow T \ \{I.bt = T.bt;\ I.w = T.w\} \ I\ ; \ \{Enter\_In\_Symtab(I.nm, I.dt, I.s)\}$$

$$T \rightarrow \text{int} \ \{T.bt = \text{int};\ T.w = 4\}$$

$$T \rightarrow double \ \{T.bt = double;\ T.w = 8\}$$

$$I \rightarrow id \ \{I.dt = I.bt;\ I.nm = id.nm;\ I.s = I.w\}$$

$$I \rightarrow id \ \{L.bt = I.bt\} \ L \ \{I.dt = L.dt;\ I.nm = id.nm;\ I.s = L.s \times I.w\}$$

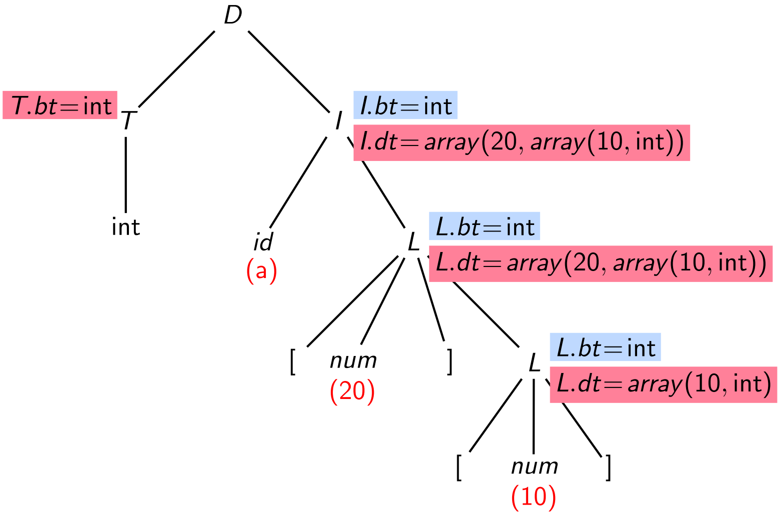
$$L \rightarrow [\ num\ ] \ \{L.dt = array(num.v, L.bt);\ L.s = num.v\}$$

$$L_1 \rightarrow [\ num\ ] \ \{L_2.bt = L_1.bt\} \ L_2 \ \{L_1.dt = array(num.v, L_2.dt);\ L_1.s = L_2.s \times num.v\}$$

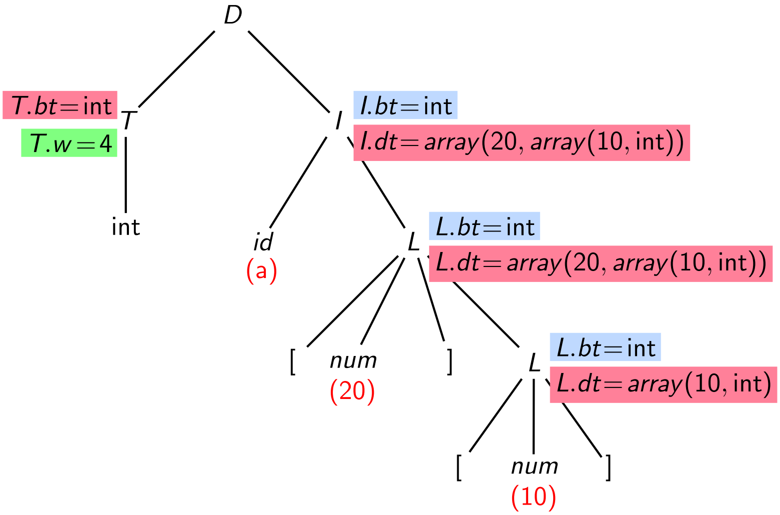
Demo of Processing C Array Declarations

- yacc script: c-decl-arrays-sdts.y
- lex script: c-decl-scanner.l

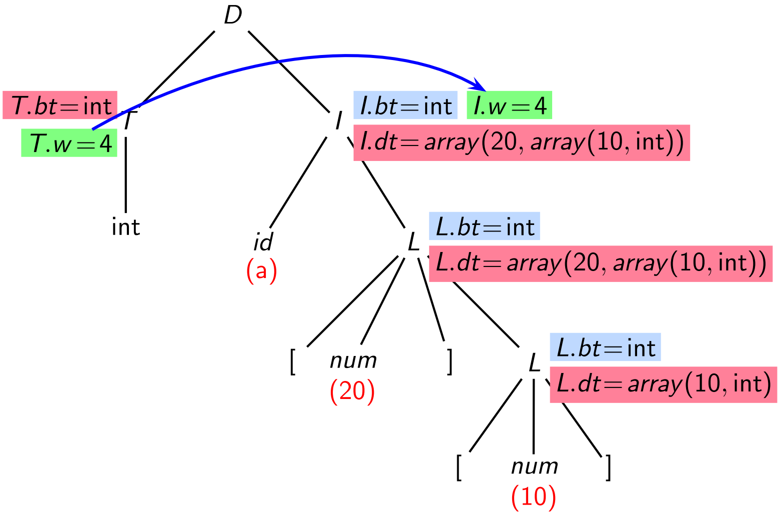
SDTS for Processing C Array Declarations: Adding Size Calculations



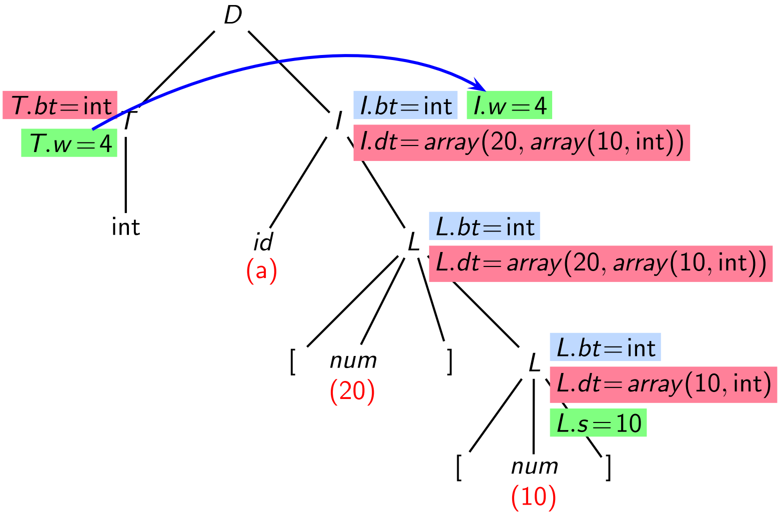
SDTS for Processing C Array Declarations: Adding Size Calculations



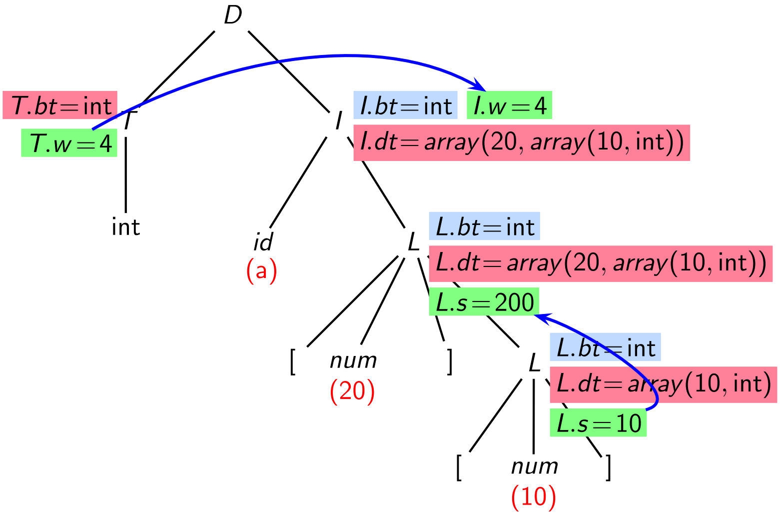
SDTS for Processing C Array Declarations: Adding Size Calculations



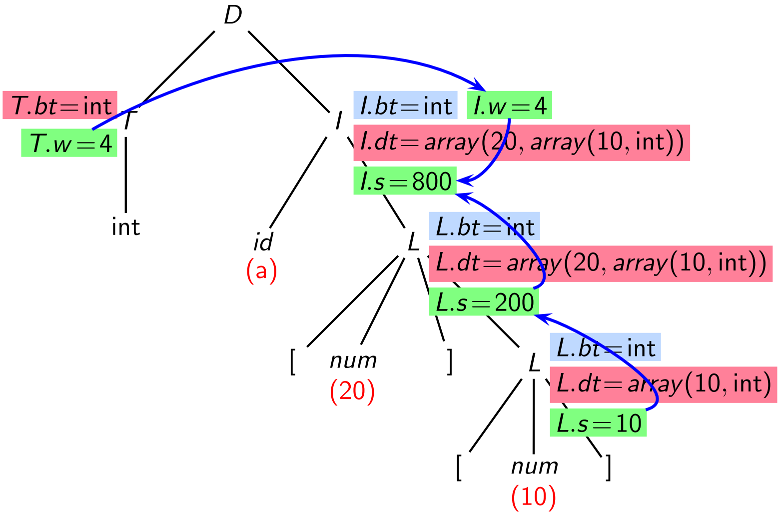
SDTS for Processing C Array Declarations: Adding Size Calculations



SDTS for Processing C Array Declarations: Adding Size Calculations



SDTS for Processing C Array Declarations: Adding Size Calculations



Including Pointers in C Array Declarations

$$
\begin{array}{rcl}
\mathit{Item} & \rightarrow & \mathit{id} \\
& | & \mathit{id}\ \mathit{ListDim} \\
& | & \mathit{ListStar}\ \mathit{id} \\
& | & \mathit{ListStar}\ \mathit{id}\ \mathit{ListDim} \\
& | & (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim} \\
& | & \mathit{ListStar}\ (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ListStar} & \rightarrow & * \\
& | & *\ \mathit{ListStar}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ListDim} & \rightarrow & [\ \mathit{num}\ ] \\
& | & [\ \mathit{num}\ ]\ \mathit{ListDim}
\end{array}
$$

Including Pointers in C Array Declarations

$$
\begin{array}{rcl}
\mathit{Item} & \rightarrow & \mathit{id} \\
& \mid & \mathit{id}\ \mathit{ListDim} \\
& \mid & \mathit{ListStar}\ \mathit{id} \\
& \mid & \mathit{ListStar}\ \mathit{id}\ \mathit{ListDim} \\
& \mid & (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim} \\
& \mid & \mathit{ListStar}\ (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ListStar} & \rightarrow & * \\
& \mid & *\ \mathit{ListStar}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ListDim} & \rightarrow & [\ \mathit{num}\ ] \\
& \mid & [\ \mathit{num}\ ]\ \mathit{ListDim}
\end{array}
$$

Including Pointers in C Array Declarations

$$
\begin{aligned}
\textit{Item} &\rightarrow \textit{id} \\
&\mid\ \textit{id ListDim} \\
&\mid\ \textit{ListStar id} \\
&\mid\ \textit{ListStar id ListDim} \\
&\mid\ (\ \textit{ListStar id}\ )\ \textit{ListDim} \\
&\mid\ \textit{ListStar}\ (\ \textit{ListStar id}\ )\ \textit{ListDim}
\end{aligned}
$$

$$
\begin{aligned}
\textit{ListStar} &\rightarrow * \\
&\mid\ *\ \textit{ListStar}
\end{aligned}
$$

$$
\begin{aligned}
\textit{ListDim} &\rightarrow [\ \textit{num}\ ] \\
&\mid\ [\ \textit{num}\ ]\ \textit{ListDim}
\end{aligned}
$$

Including Pointers in C Array Declarations

$$
\begin{aligned}
\mathit{Item} \rightarrow\ & \mathit{id} \\
\mid\ & \mathit{id}\ \mathit{ListDim} \\
\mid\ & \mathit{ListStar}\ \mathit{id} \\
\mid\ & \mathit{ListStar}\ \mathit{id}\ \mathit{ListDim} \\
\mid\ & (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim} \\
\mid\ & \mathit{ListStar}\ (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{ListStar} \rightarrow\ & * \\
\mid\ & *\ \mathit{ListStar}
\end{aligned}
$$

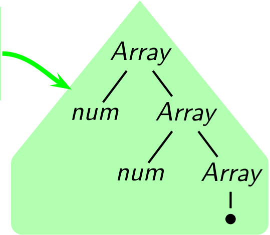
$$
\begin{aligned}
\mathit{ListDim} \rightarrow\ & [\ \mathit{num}\ ] \\
\mid\ & [\ \mathit{num}\ ]\ \mathit{ListDim}
\end{aligned}
$$

Including Pointers in C Array Declarations

$$\begin{array}{rcl} \mathit{Item} & \rightarrow & \mathit{id} \\ & \mid & \mathit{id}\ \mathit{ListDim} \\ & \mid & \mathit{ListStar}\ \mathit{id} \\ & \mid & \mathit{ListStar}\ \mathit{id}\ \mathit{ListDim} \\ & \mid & (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim} \\ & \mid & \mathit{ListStar}\ (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim} \end{array}$$

$$\begin{array}{rcl} \mathit{ListStar} & \rightarrow & * \\ & \mid & *\ \mathit{ListStar} \end{array}$$

$$\begin{array}{rcl} \mathit{ListDim} & \rightarrow & [\ \mathit{num}\ ] \\ & \mid & [\ \mathit{num}\ ]\ \mathit{ListDim} \end{array}$$

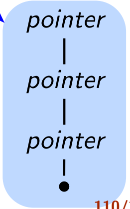


Including Pointers in C Array Declarations

$$\begin{array}{rcl}
\mathit{Item} & \rightarrow & \mathit{id} \\
& \mid & \mathit{id}\ \mathit{ListDim} \\
& \mid & \mathit{ListStar}\ \mathit{id} \\
& \mid & \mathit{ListStar}\ \mathit{id}\ \mathit{ListDim} \\
& \mid & (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim} \\
& \mid & \mathit{ListStar}\ (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim}
\end{array}$$

$$\begin{array}{rcl}
\mathit{ListStar} & \rightarrow & * \\
& \mid & *\ \mathit{ListStar}
\end{array}$$

$$\begin{array}{rcl}
\mathit{ListDim} & \rightarrow & [\ \mathit{num}\ ] \\
& \mid & [\ \mathit{num}\ ]\ \mathit{ListDim}
\end{array}$$

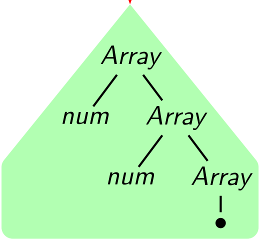


Including Pointers in C Array Declarations

$$
\begin{array}{rcl}
\textit{Item} & \rightarrow & \textit{id} \\
& | & \textit{id ListDim} \\
& | & \textit{ListStar id} \\
& | & \textit{ListStar id ListDim} \\
& | & (\ \textit{ListStar id}\ )\ \textit{ListDim} \\
& | & \textit{ListStar}\ (\ \textit{ListStar id}\ )\ \textit{ListDim}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{ListStar} & \rightarrow & * \\
& | & *\ \textit{ListStar}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{ListDim} & \rightarrow & [\ \textit{num}\ ] \\
& | & [\ \textit{num}\ ]\ \textit{ListDim}
\end{array}
$$



Including Pointers in C Array Declarations

$$
\begin{array}{lcl}
\mathit{Item} & \rightarrow & \mathit{id} \\
& \mid & \mathit{id}\ \mathit{ListDim} \\
& \mid & \mathit{ListStar}\ \mathit{id} \\
& \mid & \mathit{ListStar}\ \mathit{id}\ \mathit{ListDim} \\
& \mid & (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim} \\
& \mid & \mathit{ListStar}\ (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{ListStar} & \rightarrow & * \\
& \mid & *\ \mathit{ListStar}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{ListDim} & \rightarrow & [\ \mathit{num}\ ] \\
& \mid & [\ \mathit{num}\ ]\ \mathit{ListDim}
\end{array}
$$

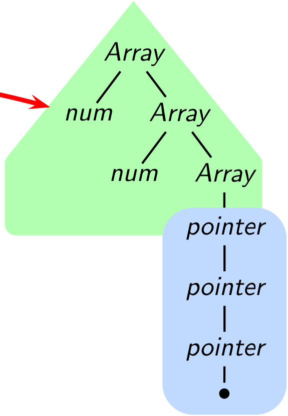
pointer
|
pointer
|
pointer
|
•

Including Pointers in C Array Declarations

$$
\begin{array}{rcl}
\mathit{Item} & \rightarrow & \mathit{id} \\
& | & \mathit{id}\ \mathit{ListDim} \\
& | & \mathit{ListStar}\ \mathit{id} \\
& | & \mathit{ListStar}\ \mathit{id}\ \mathit{ListDim} \\
& | & (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim} \\
& | & \mathit{ListStar}\ (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ListStar} & \rightarrow & * \\
& | & *\ \mathit{ListStar}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ListDim} & \rightarrow & [\ \mathit{num}\ ] \\
& | & [\ \mathit{num}\ ]\ \mathit{ListDim}
\end{array}
$$

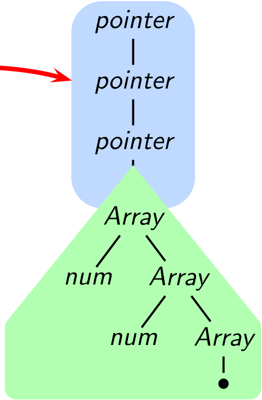


Including Pointers in C Array Declarations

$$
\begin{array}{rcl}
\textit{Item} & \rightarrow & \textit{id} \\
& \mid & \textit{id ListDim} \\
& \mid & \textit{ListStar id} \\
& \mid & \textit{ListStar id ListDim} \\
& \mid & (\ \textit{ListStar id}\ )\ \textit{ListDim} \\
& \mid & \textit{ListStar}\ (\ \textit{ListStar id}\ )\ \textit{ListDim}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{ListStar} & \rightarrow & * \\
& \mid & *\ \textit{ListStar}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{ListDim} & \rightarrow & [\ \textit{num}\ ] \\
& \mid & [\ \textit{num}\ ]\ \textit{ListDim}
\end{array}
$$

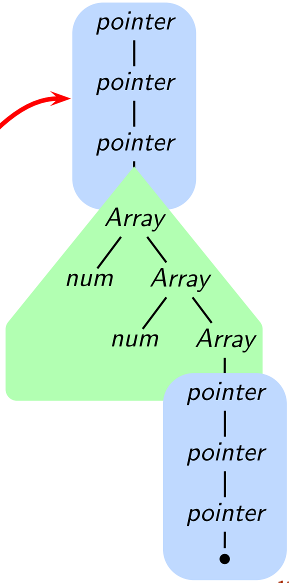


Including Pointers in C Array Declarations

$$
\begin{array}{rcl}
\mathit{Item} & \rightarrow & \mathit{id} \\
& \mid & \mathit{id}\ \mathit{ListDim} \\
& \mid & \mathit{ListStar}\ \mathit{id} \\
& \mid & \mathit{ListStar}\ \mathit{id}\ \mathit{ListDim} \\
& \mid & (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim} \\
& \mid & \mathit{ListStar}\ (\ \mathit{ListStar}\ \mathit{id}\ )\ \mathit{ListDim}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ListStar} & \rightarrow & * \\
& \mid & *\ \mathit{ListStar}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ListDim} & \rightarrow & [\ \mathit{num}\ ] \\
& \mid & [\ \mathit{num}\ ]\ \mathit{ListDim}
\end{array}
$$



Adding a List

```
int *a[10][20], **b, c;
```

$$\begin{aligned}
Decl &\rightarrow \mathrm{T}\ List\ ; \\
List_1 &\rightarrow \{List_2.bt = List_1.bt\}\ \ List_2\ \ , \\
&\ \ \{Item.bt = List_1.bt\}\ \ Item \\
List &\rightarrow \{Item.bt = List.bt\}\ \ Item
\end{aligned}$$

Adding a List

```
int *a[10][20], **b, c;
```

$$Decl \rightarrow \text{T } List \;\; ;$$

$$List_1 \rightarrow \{List_2.bt = List_1.bt\} \;\; List_2 \;\; , \;\; \{Item.bt = List_1.bt\} \;\; Item$$

$$List \rightarrow \{Item.bt = List.bt\} \;\; Item$$

$$List_1 \rightarrow \$ACT1 \;\; List_2 \;\; , \;\; \$ACT2 \;\; Item$$

$$List \rightarrow \$ACT3 \;\; Item$$

$$\$ACT1 \rightarrow \%empty \;\; \{List_2.bt = List_1.bt\}$$

$$\$ACT2 \rightarrow \%empty \;\; \{Item.bt = List_1.bt\}$$

$$\$ACT3 \rightarrow \%empty \;\; \{Item.bt = List.bt\}$$

Adding a List

```
int *a[10][20], **b, c;
```

$$Decl \rightarrow \mathrm{T}\ List\ ;$$

$$List_1 \rightarrow \{List_2.bt = List_1.bt\}\ List_2\ ,$$
$$\{Item.bt = List_1.bt\}\ Item$$

$$List \rightarrow \{Item.bt = List.bt\}\ Item$$

$$List_1 \rightarrow \$ACT1\ List_2\ ,\ \$ACT2\ Item$$

$$List \rightarrow \$ACT3\ Item$$

$$\$ACT1 \rightarrow \%empty\ \{List_2.bt = List_1.bt\}$$

$$\$ACT2 \rightarrow \%empty\ \{Item.bt = List_1.bt\}$$

$$\$ACT3 \rightarrow \%empty\ \{Item.bt = List.bt\}$$

The actions in the beginning of the RHSs give rise to reduce-reduce conflict in a yacc/bison parser

Adding A List

```
int *a[10][20], **b, c;
```

$$\begin{aligned}
Decl &\rightarrow \mathrm{T}\ List\ \ ; \\
List_1 &\rightarrow \{List_2.bt = List_1.bt\}\ \ List_2\ \ , \\
&\qquad \{Item.bt = List_1.bt\}\ \ Item \\
List &\rightarrow \{Item.bt = List.bt\}\ \ Item
\end{aligned}$$

Adding A List

```
int *a[10][20], **b, c;
```

$$Decl \rightarrow \text{T } List \ ;$$

$$List_1 \rightarrow \{List_2.bt = List_1.bt\} \ List_2 \ ,$$
$$\{Item.bt = List_1.bt\} \ Item$$

$$List \rightarrow \{Item.bt = List.bt\} \ Item$$

$$Decl \rightarrow \text{T } List \ ;$$

$$List \rightarrow \{Item.bt = List.bt\} \ Item$$
$$\{List\_Tail.bt = List.bt\} \ List\_Tail$$

$$List\_Tail \rightarrow \ , \ \{List.bt = List\_Tail.bt\} \ List$$

$$List\_Tail \rightarrow \%empty$$

No reduce-reduce conflicts because recursion on *List* is an indirect recursion rather than a direct recursion, separating the two marker non-terminals representing the action before *Item*, apart

Demo of Processing C Array Declarations with Pointers

- Parser (without attribute evaluation)
  - yacc script: c-decl-processing-grammar.y
  - lex script: c-decl-scanner-without-actions.l
- SDTS
  - yacc script: c-decl-arrays-pointers-sdts.y
  - lex script: c-decl-scanner.l