Abstract Syntax Tree and Symbol Tables

Fan Long

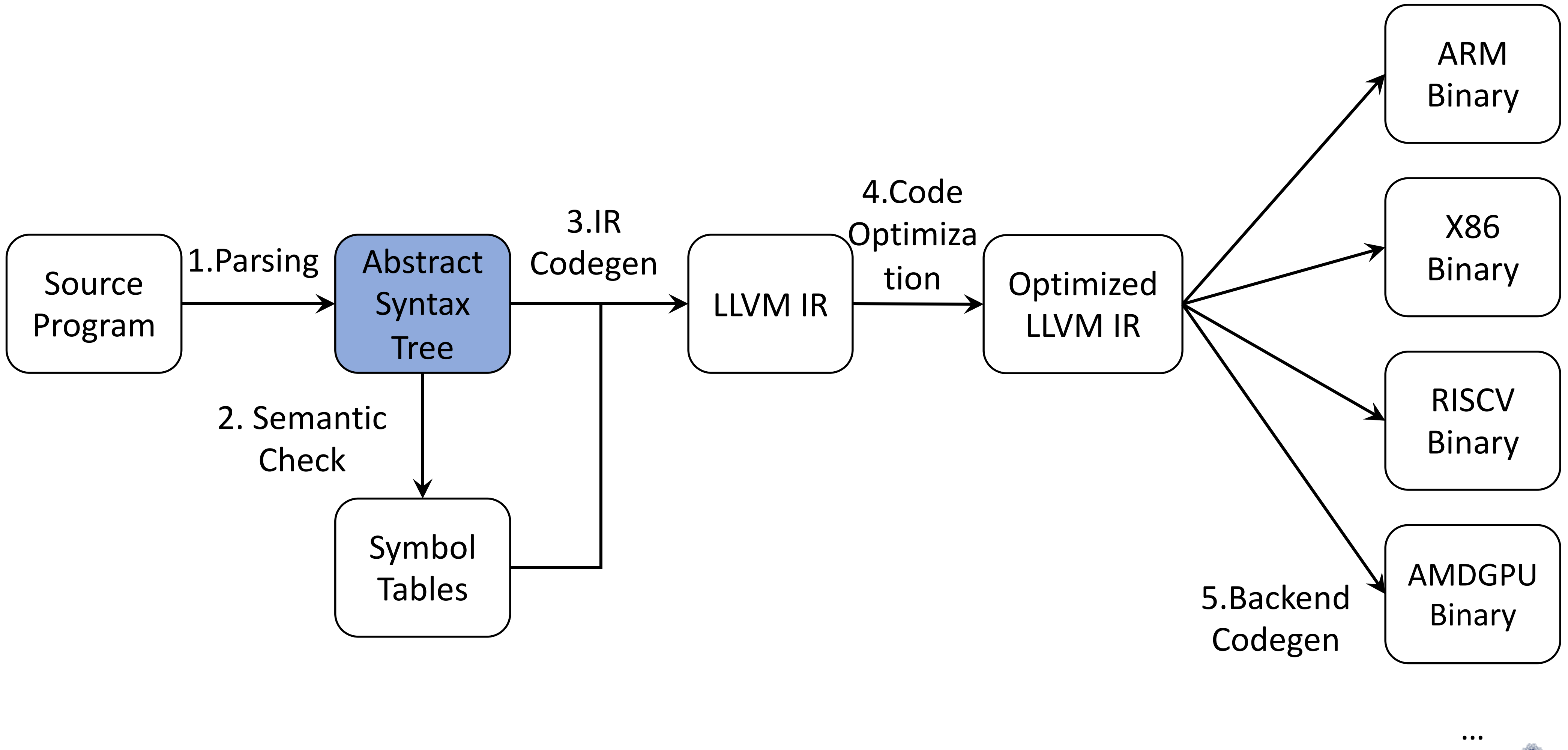
University of Toronto

Internal Compiler Data Structures

- **Parse Tree**: the output of the syntax analysis. It represents the syntax structure of the program.
- **Abstract Syntax Tree**: a modified parse tree. In most compilers, it shares the same structure as the original parse tree but describes essential program structure.
- **Symbol Tables**: Data structures to maintain semantic and code generation information for variable/function/type identifiers.
- **Intermediate Representation (IR)**: Architecture independent code that is close to assemblies, .e.g., LLVM IR. It serves as a platform for complicated compiler optimizations. Simple compilers may choose to directly generate assembly from AST.

UNIVERSITY OF TORONTO

Typical LLVM Compiler Workflow



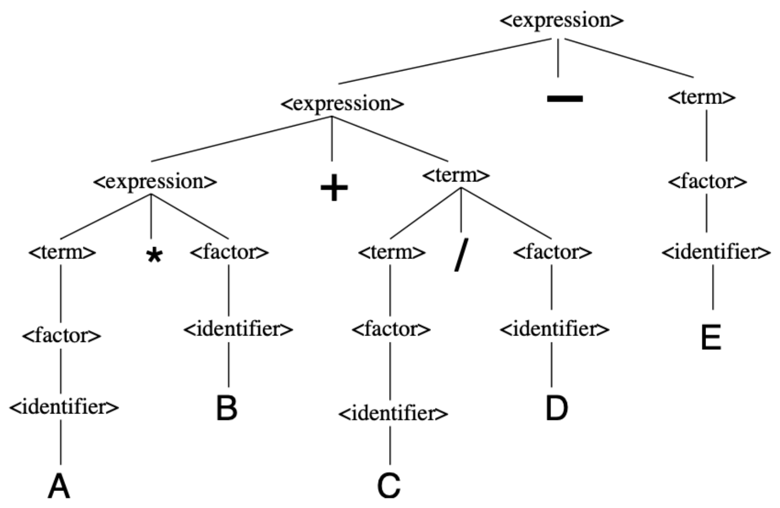
Parse Tree v.s. Abstract Syntax Tree

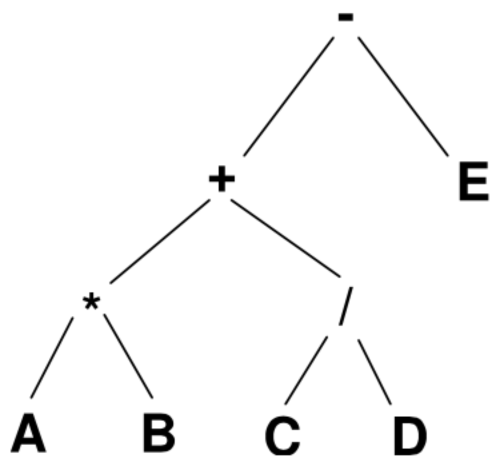
- **Parse Tree**:

Complete representation of the syntactic structure of the program according to some grammar.

- **AST**:

Similar to parse tree but only describes essential program structure.



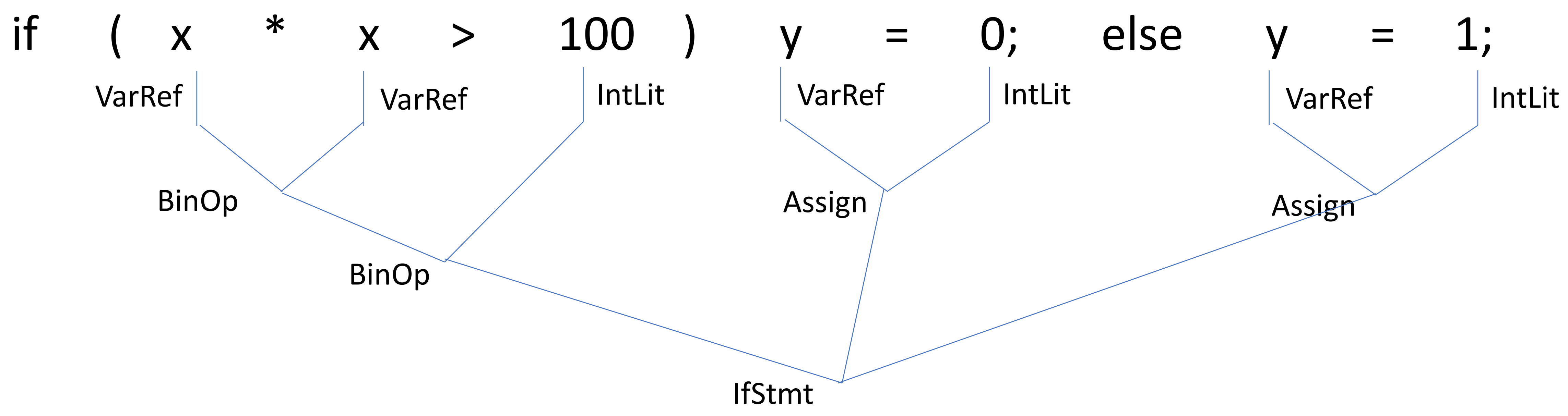


Abstract Syntax Tree

- Abstract Syntax Trees (ASTs) are designed to capture all the essential structural information about a program being compiled.
- The basic process for *building* an AST is very simple:
  - The leaf nodes are typically constants and identifiers. Build a node to represent each of these entities.
  - Interior nodes are build as needed to represent language constructs. Typically, interior nodes contain links to one or more child nodes.
- The *processing* of AST nodes for semantic analysis is also simple:
  - Process the AST *depth first*. This guarantees that child nodes are processed before parent nodes.
  - At each parent node do any processing required using information from already processed child nodes.

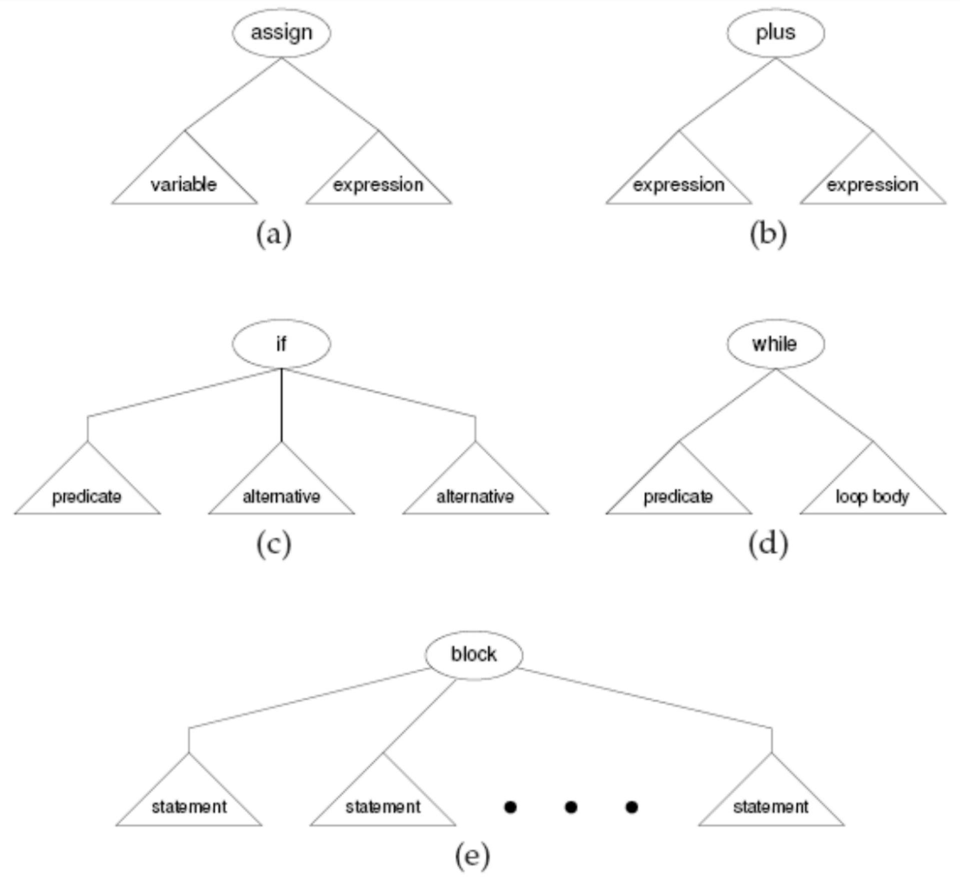
UNIVERSITY OF TORONTO

Abstract Syntax Tree Example



ASTs are Language Specific

- Typically AST notes are custom designed for each of the major constructs in the language.



A Very General Abstract Syntax Tree Node

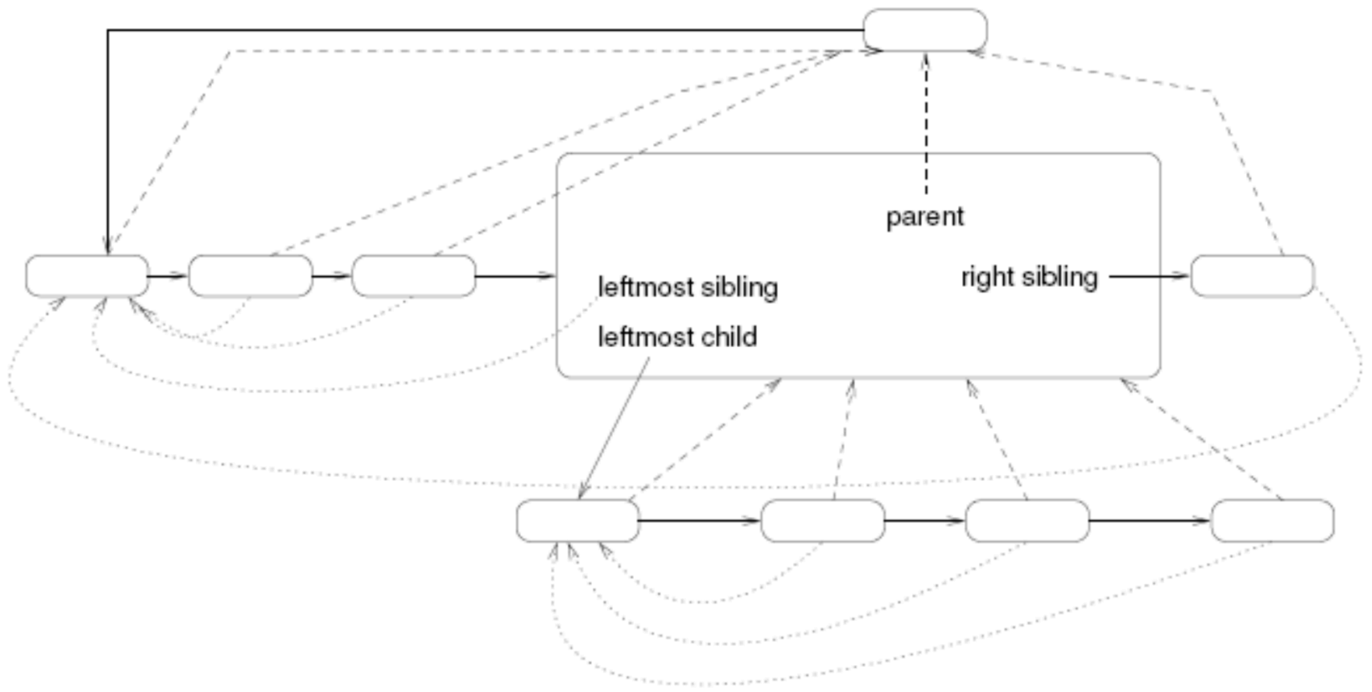


Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

Generic ASTNode Class in MiniC Compiler

```
struct SourceLocation {
  int Line, Row;
}
class ASTNode {
    std::vector<ASTNode*> Children;
    ASTNode* Parent;
    SourceLocation SrcLoc;
    Program* Root;
}
```

- All other AST classes in MiniC Compiler inherit this base class.
- Children nodes will have different semantic meanings based on the specific AST class type.

IfStatment Class in MiniC Compiler

```
class Statement : public ASTNode { };
class IfStatement : public Statement {
public:
  Expr* condExpr() {
    return (Expr*) getChild(0); }
  Statement* thenStmt() {
    return (Statement*) getChild(1); }
  bool hasElse() {
    return numChildren() > 2; }
  Statement* elseStmt() {
    assert( numChildren() > 2);
    return (Statement*) getChild(2); }
};
```

- The children are the conditional expression, the then body statement, and the else body statement, respectively.
- Expr, Statement, and IfStatement are all sub-class of ASTNode.

CallExpr Class in MiniC Compiler

```
class Expr : public ASTNode {
  Type ExprType; }
class CallExpr : public Expr {
public:
  Identifier* callee() {
    return (Identifier*) getChild(0);
  }
  size_t numArgs() {
    assert( numChildren() > 0);
    return numChildren() - 1;
  }
  Expr *arg(size_t i) {
    assert ( i + 1 < numChildren());
    return (Expr*)getChild(i + 1);
  }
};
```

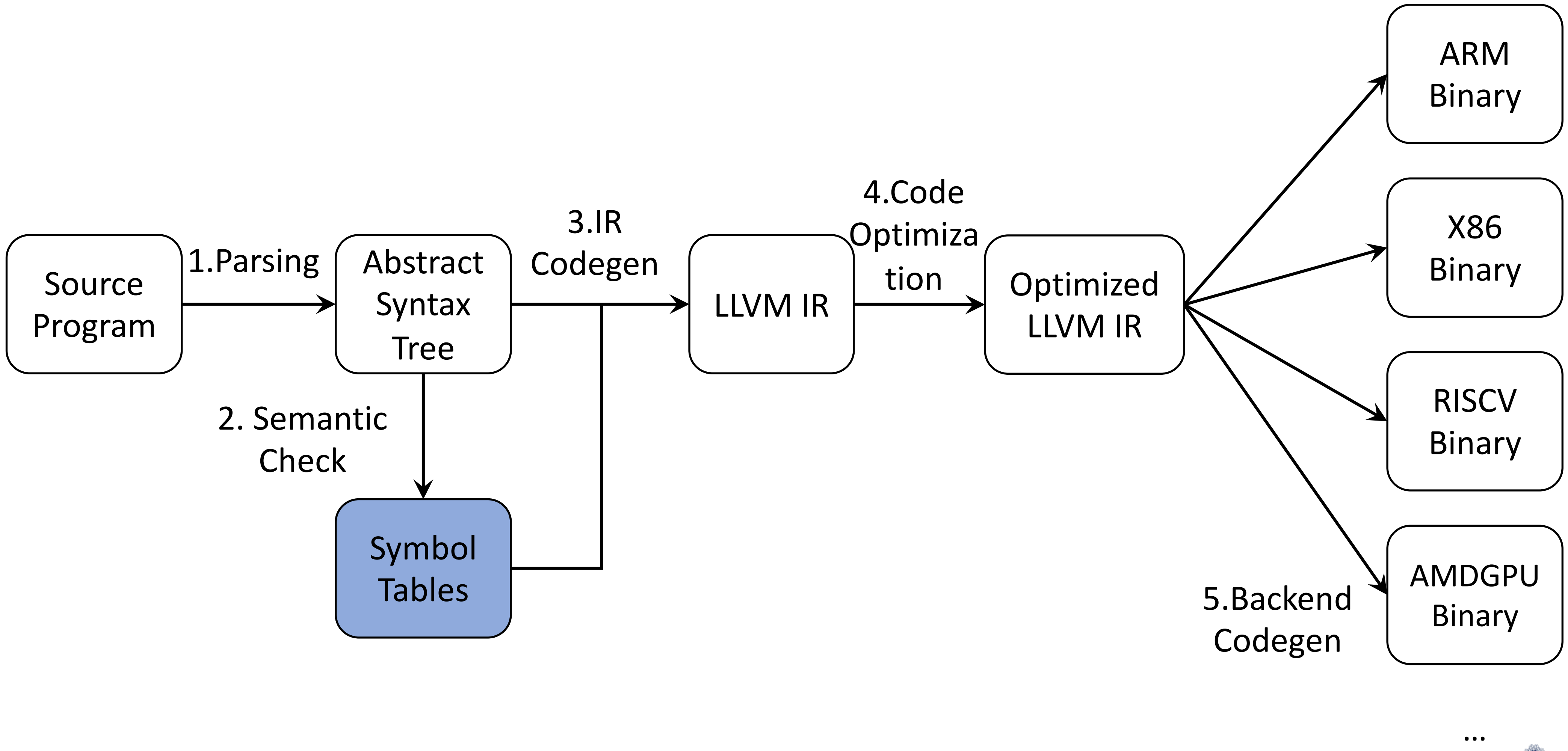
- The first child is the callee.
- The follow up children are expressions that correspond to the passed arguments in order.

How to Build MiniC Compiler AST with ANTLR

- Develop compiler grammar rules for MiniC. (Assignment 2)
- Attach compiler actions to each grammar rule.

- The actions typical perform the following tasks:
  - Allocate a new AST class based on the type of the processing grammar.
  - Parse sub-grammars (non-terminals) and collect the resulting AST node result.
  - Push the resulting AST nodes as the children of the new AST class in order.
  - Collect necessary information for diagnosis such as source locations.

Typical LLVM Compiler Workflow



Compiler Tables

- The *Symbol Table* contains information about declared things (usually identifiers).
- A *Type Table* is used to record information about builtin and user declared types. Necessary for languages that allow arbitrary user declared types.
- Efficient table management is often a major performance issue in the design of a compiler. Table search is the dominant operation.
- Table lookup must follow the languages scope rules. Table usage:
  - Declaration processing - add entries for identifiers and types being declared
  - Semantic analysis - look up identifiers to determine their attributes. Look up types to validate program usage.
  - Code generation - look up identifiers to determine their attributes.

Compiler Tables

- Possible table organizations
  - One global table for entire program.
  - Separate linked tables for each major scope

- Table storage
  - Fixed size memory resident, limits program size
  - Statically or dynamically allocated in memory.
  - Partially memory resident, remainder on disk. LRU caching works well.

Typical Symbol Table Entries

- A symbol table entry contains the attributes of symbols that might be different for each symbol
- A typical symbol table entry might contain:
  - Name of the item
  - Kind of item, e.g. constant, variable, type, procedure, function, etc.
  - Type of the item (index into type table)
  - Attributes or properties associated with the item (usually language specific)
  - Size of the item (or derive from type)
  - Run time address for the item, e.g. base register and offset.
  - Value for the item, e.g. value of constants, initial value for variables (might be index into a table of constants)
  - Links to related symbols, e.g. parameter lists, enumerated constants, fields in a struct or union.

Symbol Table Operations

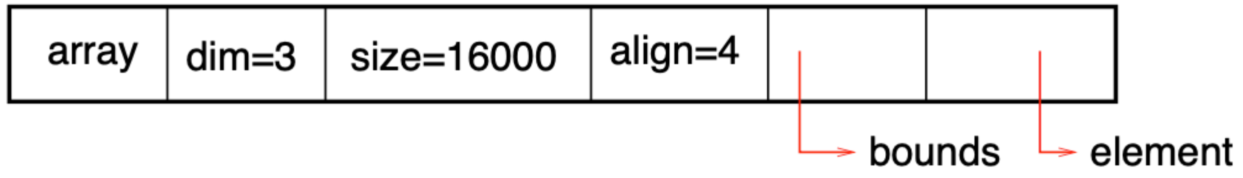
- The following operations are typical of a symbol table class/module
  - Create Symbol Table
  - Enter new scope
  - Exit Scope
  - Lookup symbol in current scope
  - Lookup symbol using scope rule
  - Enter new symbol in current scope
  - Enter new symbol in designated scope
  - Delete symbol from table
  - Retain symbol in symbol table
  - Get or set fields in the symbol table

Generic Type Table Entry

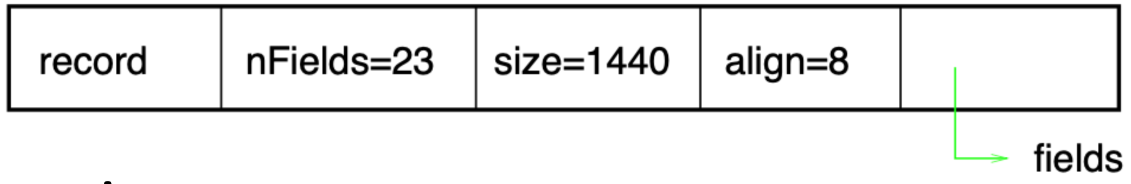
- A type table entry contains the information that might be different for each distinct type.
- A typical type table entry might contain:
  - Name of the item (often omitted)
  - Kind of type, e.g., typedef, enum, scalar, array, struct, union, etc.
  - Attributes or properties associated with the type, e.g., const, static, etc.
  - Size and memory alignment for objects of this type.
  - Actual definition of the type, usually involves links to embedded components.
  - Links to related definitions.
  - For many modern languages, the type table is a directed acyclic graph.

Generic Type Table Entry

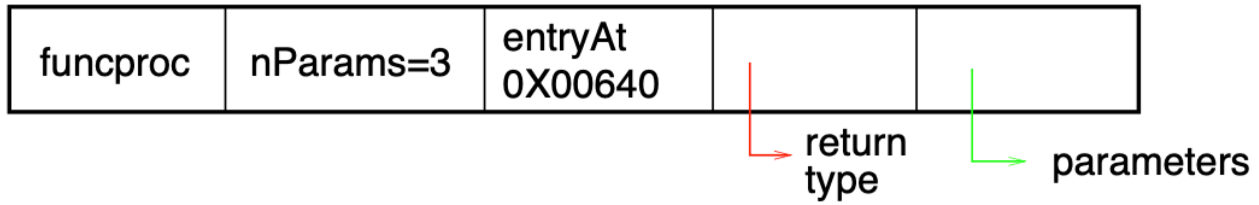
- Example: array type
  - Number of dimensions, size/alignment, bound of each dimension



- Example: struct
  - Number of fields, size/alignment, link to entries for fields



- Example: function
  - Number of parameters, entry point, links to return type and parameters

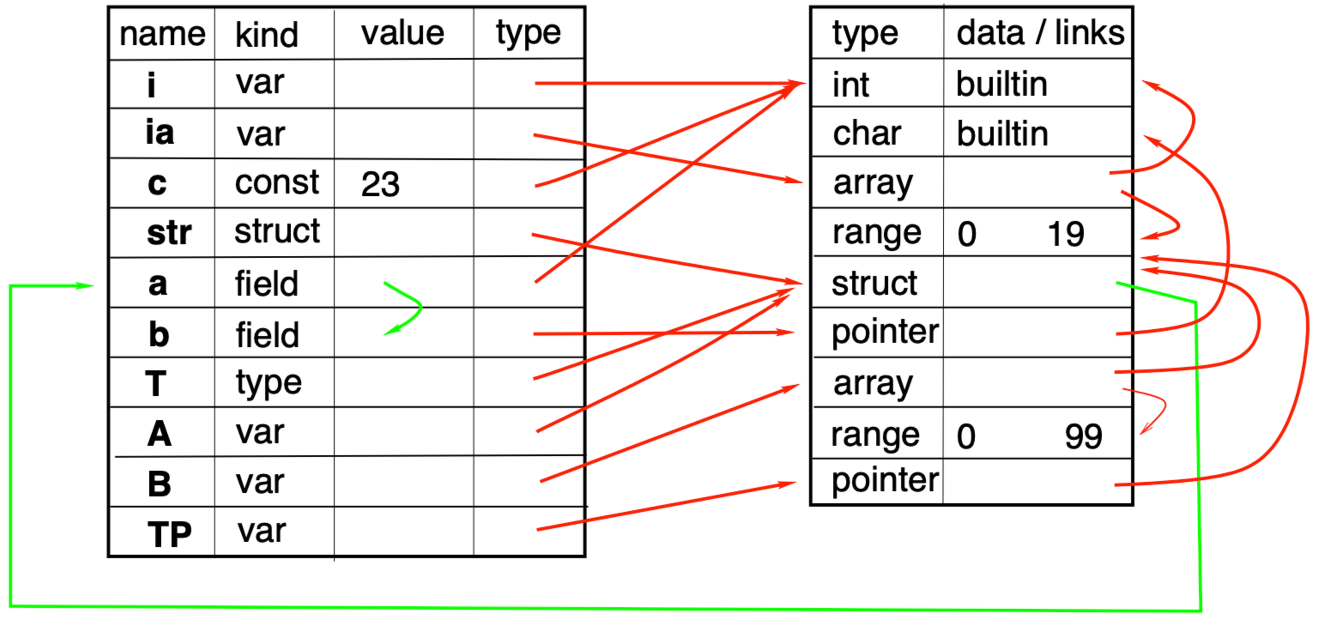


Symbol and Type Table Example

```
typedef
    struct str {
        int a ;
        char * b ;
    } T ;
```

```
int i, ia[ 20 ] ;
 const int c = 23 ;

T   A, B[ 100 ], * TP ;
```



name	kind	value	type
i	var		
ia	var		
c	const	23	
str	struct		
a	field		
b	field		
T	type		
A	var		
B	var		
TP	var		

type	data / links	
int	builtin	
char	builtin	
array		
range	0	19
struct		
pointer		
array		
range	0	99
pointer		

Handling Values

- The compiler needs to be able to represent in its tables any kind of *value*, i.e. constant, that could occur in a program.
- To conserve table space, some form of union data structure is often used:

```
struct constantDesc {
     short  constantKind ;
     union  {
            int     intValue ;
            float   floatValue ;
            char    charValue ;
            char   *stringValue ;
            void   *bigValue ;
     } constantValue ;
 }
```

- Large constant values, e.g. initialization for arrays or structs often require special storage.

Scopes and Declarations

- Assume that a program consists of some number of nested *scopes* of declarations (e.g., main program, **begin** - **end** blocks, functions and procedures).
- Identifiers are declared in one or more scopes.
- The *scope rule* for the programming language determines the visibility of of symbols declared in one scope in other scopes.
- The most common scope rule is to allow identifiers declared in a scope to be automatically visible in all contained (properly nested) scopes.
- Languages with modules, objects, packages or classes often have different visibility rules for identifier declared within those constructs. (e.g., C++ classes with **public**, **private**, and **protected** keywords)

Major and Minor Scopes

- **Major Scope:** A major scope corresponds to a construct with special significance in the programming language.
  - Examples: main program, body of a class, body of a routine.
  - A major scope has significance beyond symbol table lookup, it is also usually a unit for storage allocation.
- **Minor Scope:** A minor scope is a small scope of less significance that occurs within a major scope.
  - Example: scopes created within statements using { and } .
- **Recommended Strategy**
  - Merge the symbol table entries for minor scopes into the symbol table for the nearest enclosing major scope.
  - Visibility of identifiers declared in minor scopes is controlled by the lookup algorithm.

Scope Rules and Tables

- In most programming languages a program is partitioned into some (possibly overlapping) scopes of declaration.
- The scope rule for a language specifies the algorithm that must be used to search compiler tables.
- To implement most scope rules, each scope of declaration is *logically* a separate set of symbol table entries.
- For the most common scope rule, scopes are properly nested and the symbol table behaves in a strictly stack-like fashion.
- Most compilers distinguish symbol *visibility* from symbol *storage*. The scope rule determines which symbols are visible at any point in the program.

Retained Symbols

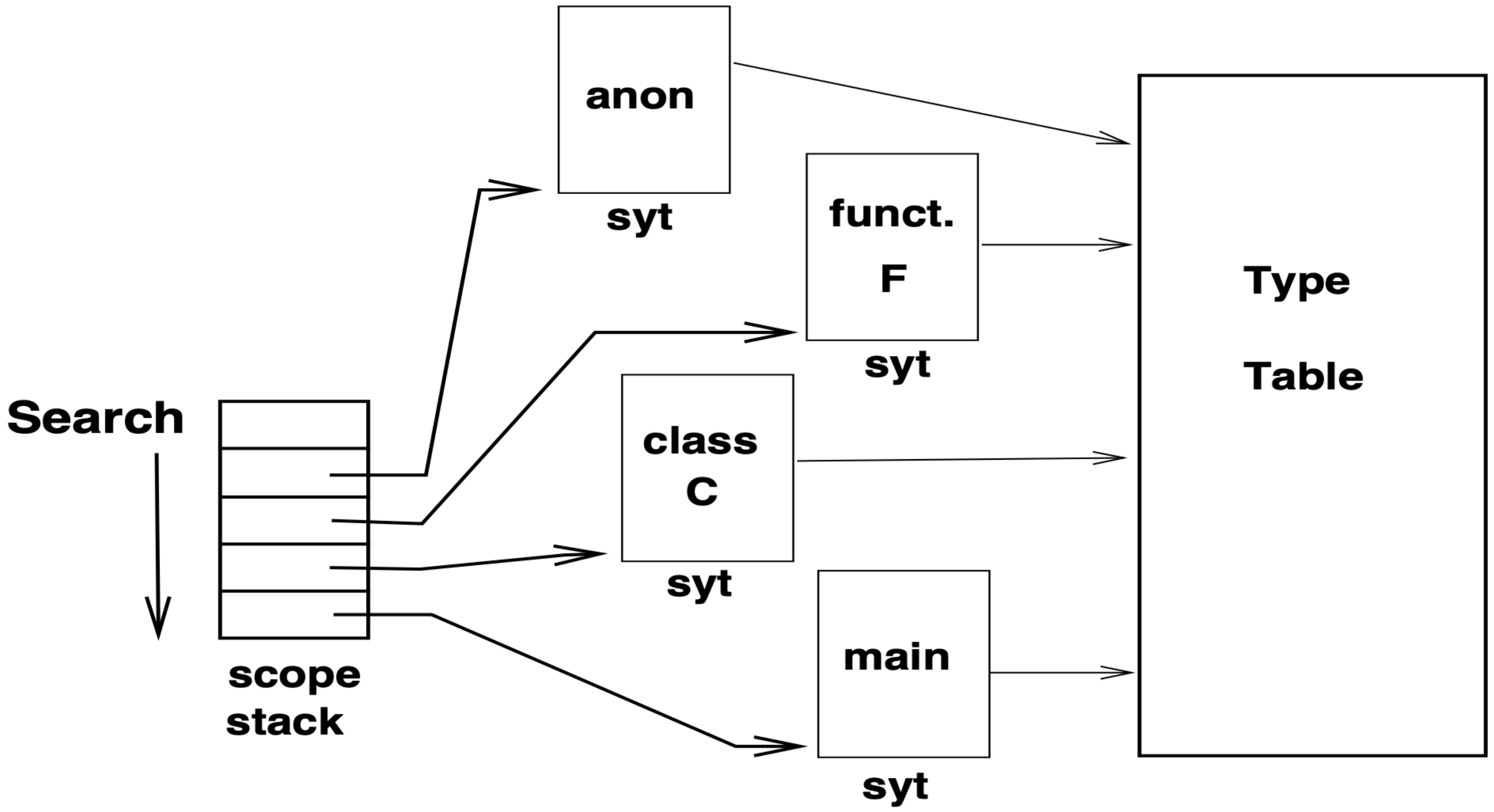
- The formal parameters of functions and procedures need special symbol table handling.
- The symbol table entries are created when the prototype or actual definition of the routine is processed.
- The entries are used during processing the body of the routine.
- After the routine has been processed, the symbol table entries for the formal parameters of the routine need to be retained in the symbol table so they can be used to process *calls* of the routine.
- The usual technique is to leave the formal parameter entries in the symbol table linked to the routine entry. The parameters won't be visible to ordinary symbol table lookup.

Symbol Table Performance Considerations

- Hash table is the most common choice for implementing the symbol table. O(1) insertion and look ups.
- For complicated compilers, map each identifier string into an index integer. Then use index integers as symbol table keys.
- **Look Up in Scopes**: Iteratively fetch the parent scope until it finds the target symbol. Or it returns null when reaching and not finding in the root program scope.
- **Cache Look Up**: It might be a good idea to cache the look up results locally so that it can return the results faster when looking for the same variable again.

UNIVERSITY OF TORONTO

Scope Stack Symbol Table Search



MiniC Symbol Table Example

```
struct VarSymbolEntry {
  Type VarType;
  llvm::Value *LLVMValue;
};
class VarSymbolTable {
  std::map<std::string, VarSymbolEntry> Table;
};
```

- Variable symbol table maps each variable to its type and the corresponding LLVM IR object (used during code generation).
- It contains methods to get/set symbol table entries for new variables.

MiniC Symbol Table Example

```
struct FuncSymbolEntry {
  Type ReturnType;
  std::vector<Type> ParameterTypes;
  bool HasBody;
};
class FuncSymbolTable {
  std::map<std::string, FuncSymbolEntry> Table;
};
```

- Variable symbol table maps each function name to its return type and the list of parameter types. It also contains a flag to indicate whether we have seen the definition (function body) yet.
- It contains methods to get/set symbol table entries for new functions.

How to Handle Symbol Table for MiniC

- MiniC does not allow forward reference. This means that we can construct the symbol table when we visit the AST tree to perform the semantic analysis.
- When encounter a valid variable declaration, add the variable with its type information into the varaible symbol table.
- When encounter a valid function declaration, add the function with its signature type information into the function symbol table.
- When checking the semantic correctness of the program, i.e., whether the type of two expressions match, you will query the symbol table accordingly.

UNIVERSITY OF TORONTO

Q/A?