Compilers (CS30003)

Lecture 15-17

Pralay Mitra

Syntax Directed Translation

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \,\|\|\, T.code \,\|\|\, '+'$

Syntax Directed Definition: A CFG together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

Synthesized Attribute: For a non-terminal *A* at a parse tree node *N* synthesized attribute is defined by a semantic rule associated with the production at *N*.

Inherited Attribute: For a non-terminal *B* at a parse tree node *N* inherited attribute is defined by a semantic rule associated with the production at the parent of *N*.

RECAP

Syntax Directed Definition

Sl No	PRODUCTION	SEMANTIC RULES
1	$L \rightarrow E \$$	L.val=E.val
2	$E \rightarrow E_1 + T$	$E.val=E_1.val + T.val$
3	$E \rightarrow T$	E.val=T.val
4	$T \rightarrow T_1 * F$	$T.val=T_1.val \times F.val$
5	$T \rightarrow F$	T.val=F.val
6	$F \rightarrow (E)$	F.val=E.val
7	$F \rightarrow$ digit	F.val=digit.lexval

RECAP

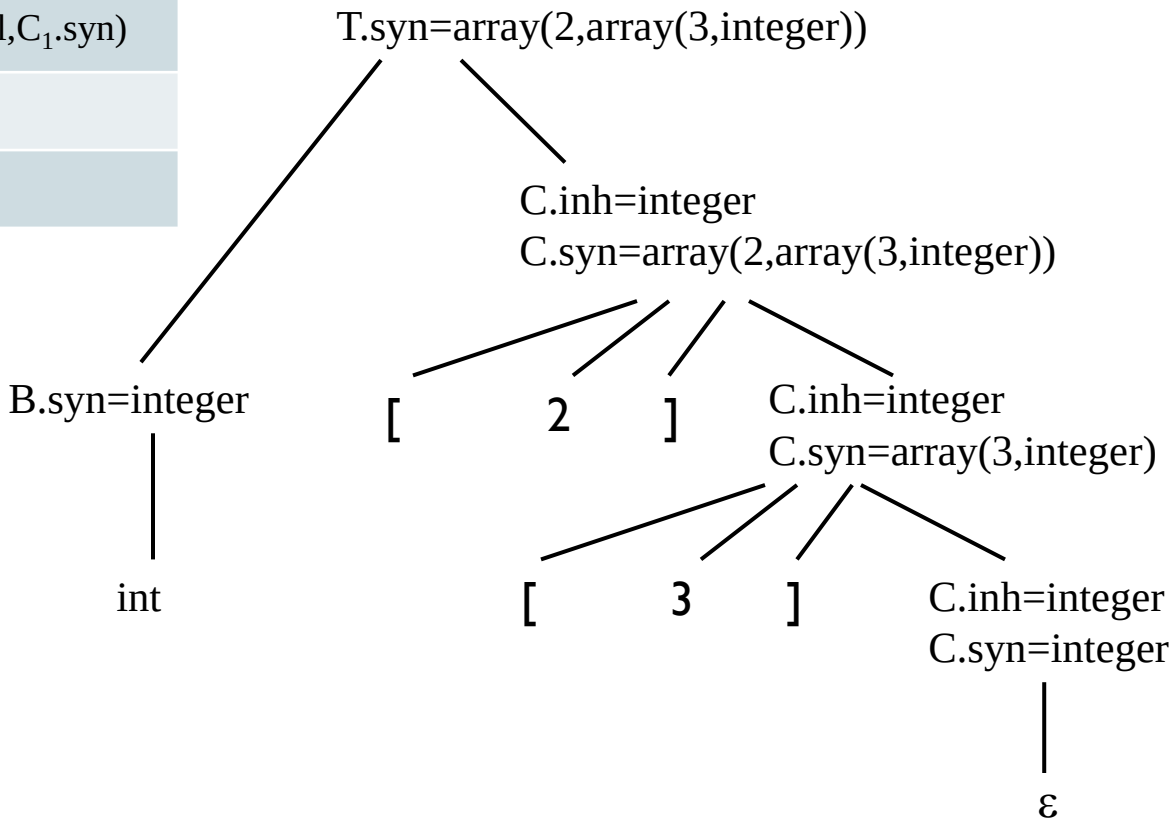
Draw an annotated parse tree for 3 * 5 + 4 $

An example

	PRODUCTION	SEMANTIC RULES
1	T → B C	T.syn=C.syn
		C.inh=B.syn
2	B → int	B.syn=integer
3	B → float	B.syn=float
4	C → [num] C_1	C.syn=array(num.val,C_1.syn)
		C_1.inh=C.inh
5	C → ε	C_1.syn=C.inh

RECAP

int [2][3]



while statement

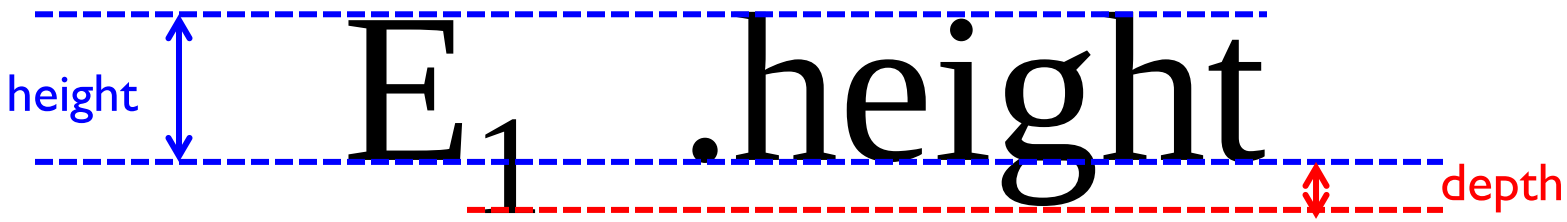
SDD

$S \rightarrow$ while (C) S_1	L1=new();
	L2=new();
	S_1.next=L1;
	C.false=S.next;
	C.true=L2;
	S.code=label || L1 || C.code || label || L2 || S_1.code

SDT

$S \rightarrow$ while (	{ L1=new(); L2=new(); C.false=S.next; C.true=L2; }
C)	{ S_1.next=L1; }
S_1	{ S.code=label || L1 || C.code || label || L2 || S_1.code; }

Type setting example



Homework

Type setting example

PRODUCTION	SEMANTIC RULES
$S \rightarrow B$	
$B \rightarrow B_1\ B_2$	
$B \rightarrow B_1$ sub B_2	
$B \rightarrow (B_1)$	
$B \rightarrow$ text	

Type setting example

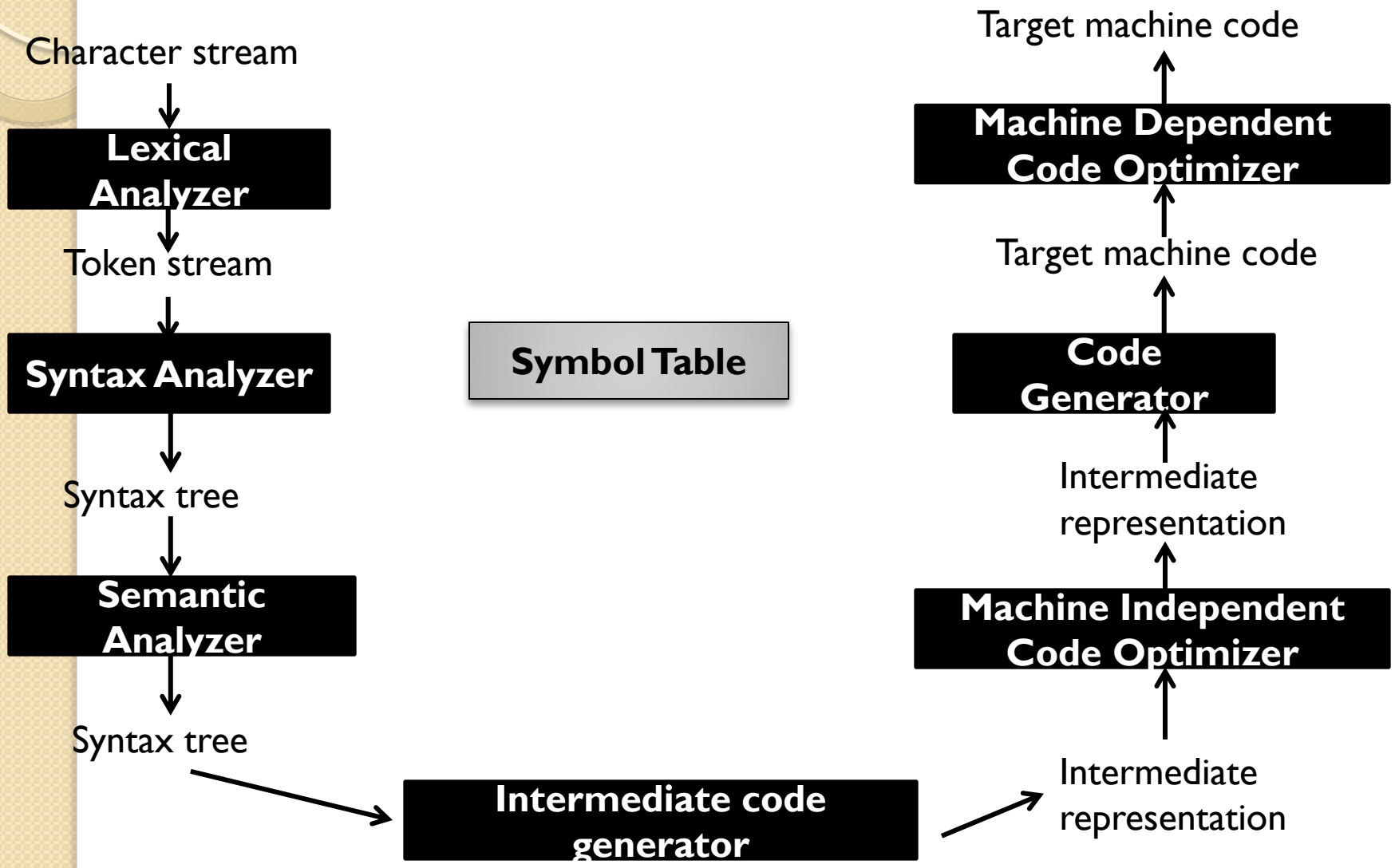
PRODUCTION	SEMANTIC RULES
S → B	B.ps=10
B → B_1 B_2	B_1.ps=B.ps
	B_2.ps=B.ps
	B.ht=max(B_1.ht,B_2.ht)
	B.dp=max(B_1.dp,B_2.dp)
B → B_1 sub B_2	B_1.ps=B.ps
	B_2.ps=0.7 × B.ps
	B.ht=max(B_1.ht,B_2.ht-0.25×B.ps)
	B.dp=max(B_1.dp,B_2.dp+0.25×B.ps)
B → (B_1)	B_1.ps=B.ps
	B.ht=B_1.ht
	B.dp=B_1.dp
B → text	B.ht=getHt(B.ps,text.lexval)
	B.dp=getDp(B.ps,text.lexval)

Practice Example

Construct an LR(0) parser for G_7:

1: $S \rightarrow A\ A$
2: $A \rightarrow a\ A$
3: $A \rightarrow b$

Phases of a compiler



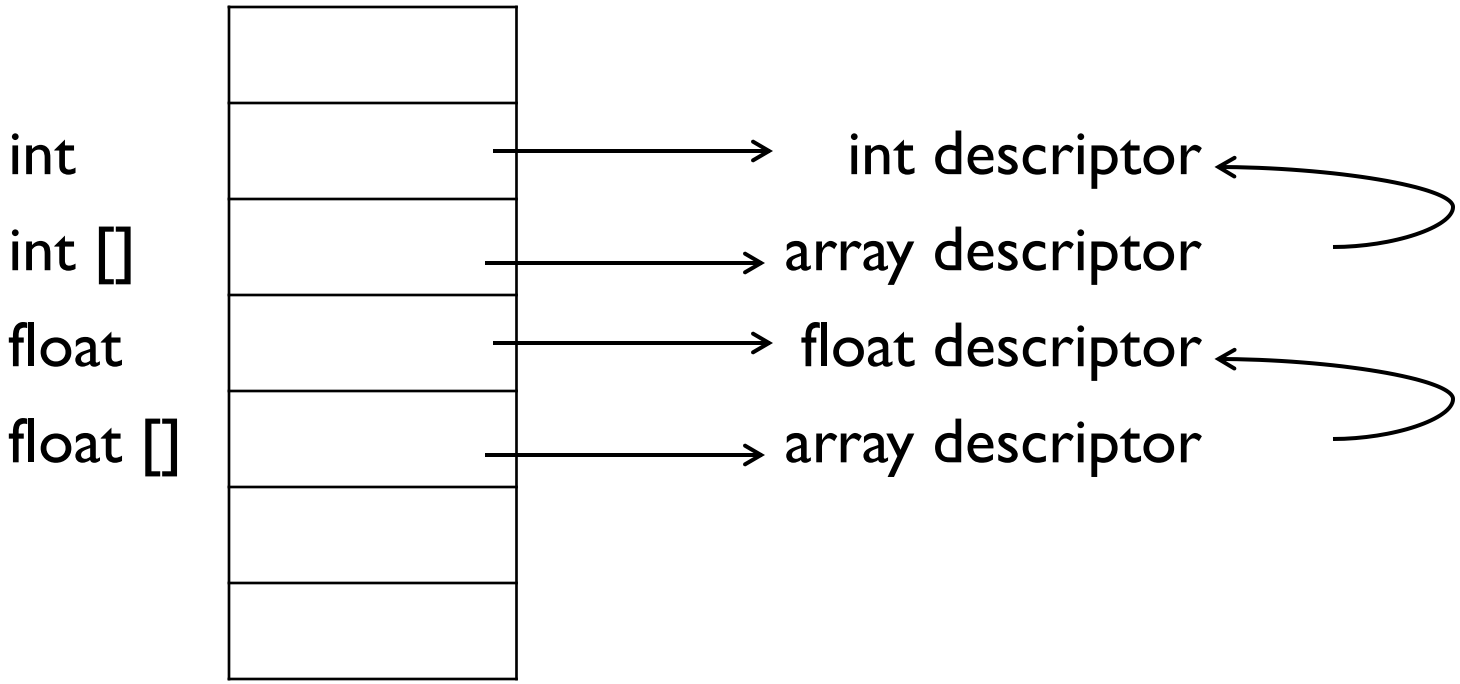
Symbol Table

- Implemented using hash table
- Helps to produce object layout in memory
- Maps identifiers to descriptors
- Basic operation
  - Look up
  - Insert
- Code to access
  - Object fields, local variables, parameters, methods

Field, Parameter, Local and Type descriptor

- Field, Parameter and Local descriptor refer to Type descriptor
  - Base type descriptors: int, float
  - Array type descriptor, which contains reference to type descriptor for array elements
- Relatively simple type descriptors
- Base type descriptor and array descriptors stored in Type Symbol Table

Type Symbol Table

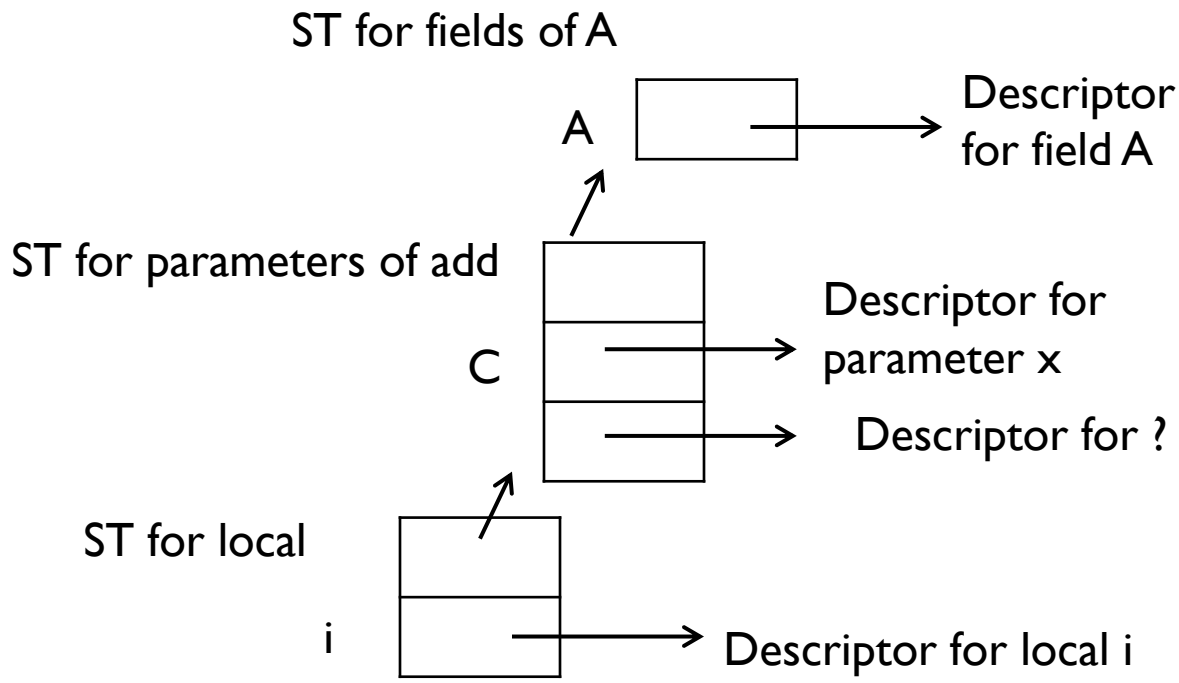


Method descriptor

- Contain reference to code for method
- Contain reference to local ST for local variables of Method
- Parent ST of Local ST is parameter ST for parameters of method

Hierarchy in ST

```
int A[];
void add(int C) {
  int i;
  i=0;
  while(i<A.length) {
     A[i]=A[i]+x;
     i=i+1;
   }
}
```



Hierarchy in ST

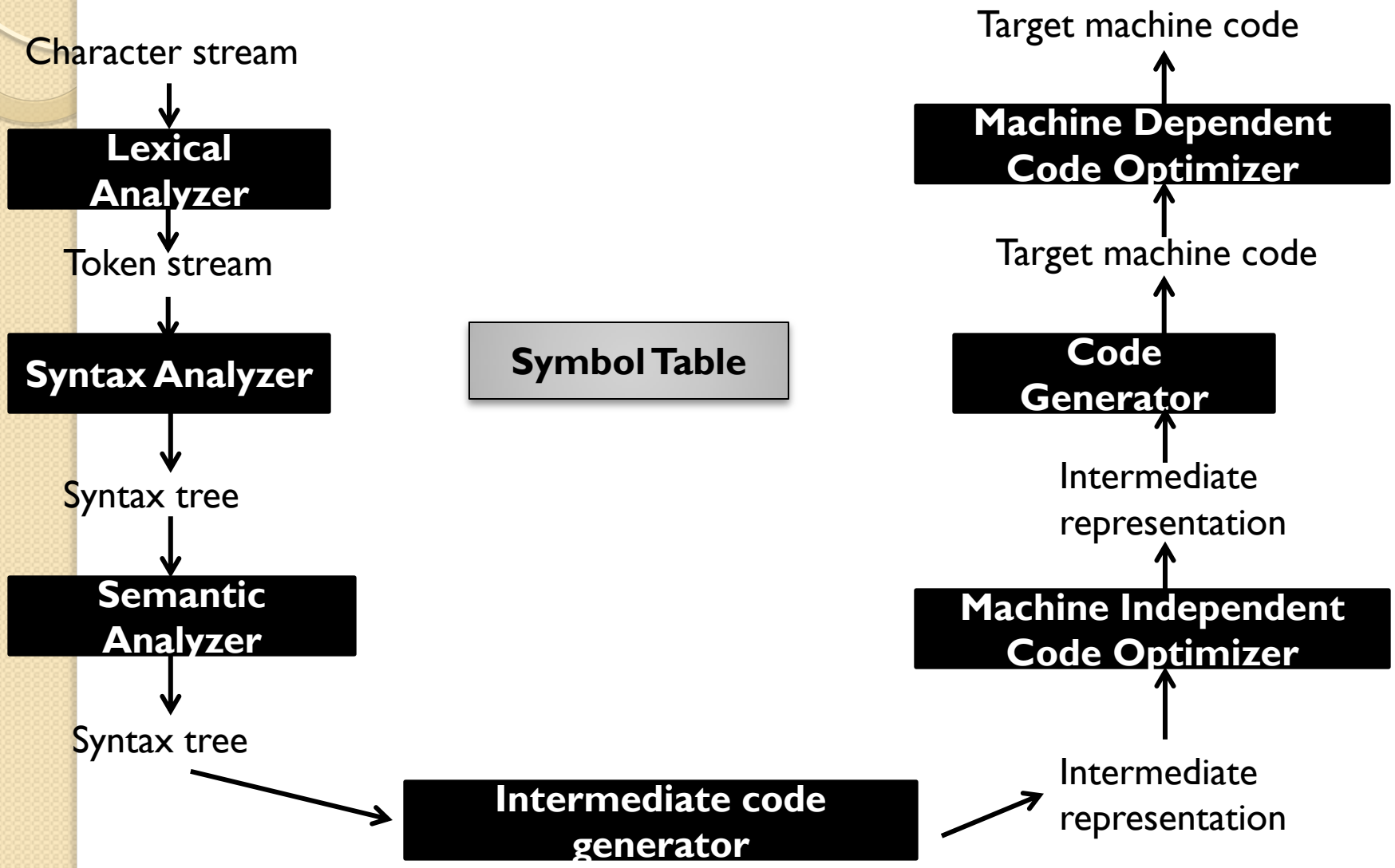
- Hierarchy in
  - Nested scopes – local scope inside field scope
  - Inheritance – child class inside parent class

- Look up proceeds up Hierarchy until descriptor is found.

Symbol Table Summary

- Program Symbol Table (Class Descriptors)
- Class Descriptors
  - Field Symbol Table (Field Descriptors)
    - Pointer to Field Symbol Table for Super Class
  - Method Symbol Table (Method Descriptors)
    - Pointer to Method Symbol Table for Superclass
- Method Descriptors
  - Local Variable Symbol Table (Local Variable Descriptors)
    - Parameter Symbol Table (Parameter Descriptors)
      - Pointer to Field Symbol Table of Receiver Class
- Local, Parameter and Field Descriptors
  - Type Descriptors in Type Symbol Table or Class Descriptors

Phases of a compiler



Parse tree – Shall I eliminate?

- Parser actions build Symbol Table
- Eliminate intermediate construction of parse tree for improved performance
- Also less code to write

Coding may be tougher compared to traversing parse tree.

Program representation goals

- Enable program analysis and transformation
  - Semantic checks, correctness checks, optimization
- Structure translation to machine code
  - Parse tree → High level IR → Low level IR → Machine code

Intermediate Representation

- High level
  - Preserves object structure
  - Preserves structured control flow
  - Helps in program analysis
- Low Level
  - Moves data model to address space
  - Eliminate structured control flow
  - Suitable for register allocation and instruction selection

High Level IR

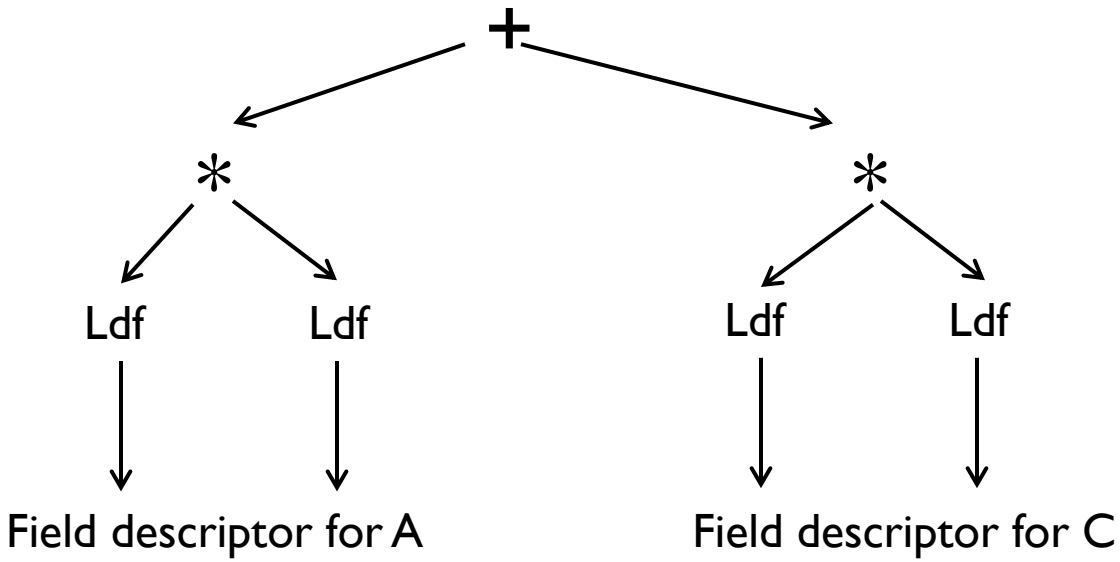
- Move towards assembly language
- Preserve high-level structure
  - Object format
  - Structured control flow
  - Distinction between parameters, locals and fields
- High-level abstractions of assembly language
  - Load and store nodes
  - Access abstract locals, parameters and fields, not memory locations directly

Expressions

- Expression trees
  - Internal nodes are operations (+, -, /)
  - Leaves are variable accesses
- Nodes
  - Ldf (field descriptor – to access field)
  - Ldl (local descriptor – to access local variable)
  - Ldp (parameter descriptor – to access parameter)
  - Lda (to access array/index)
  - Sta (to store array elements)

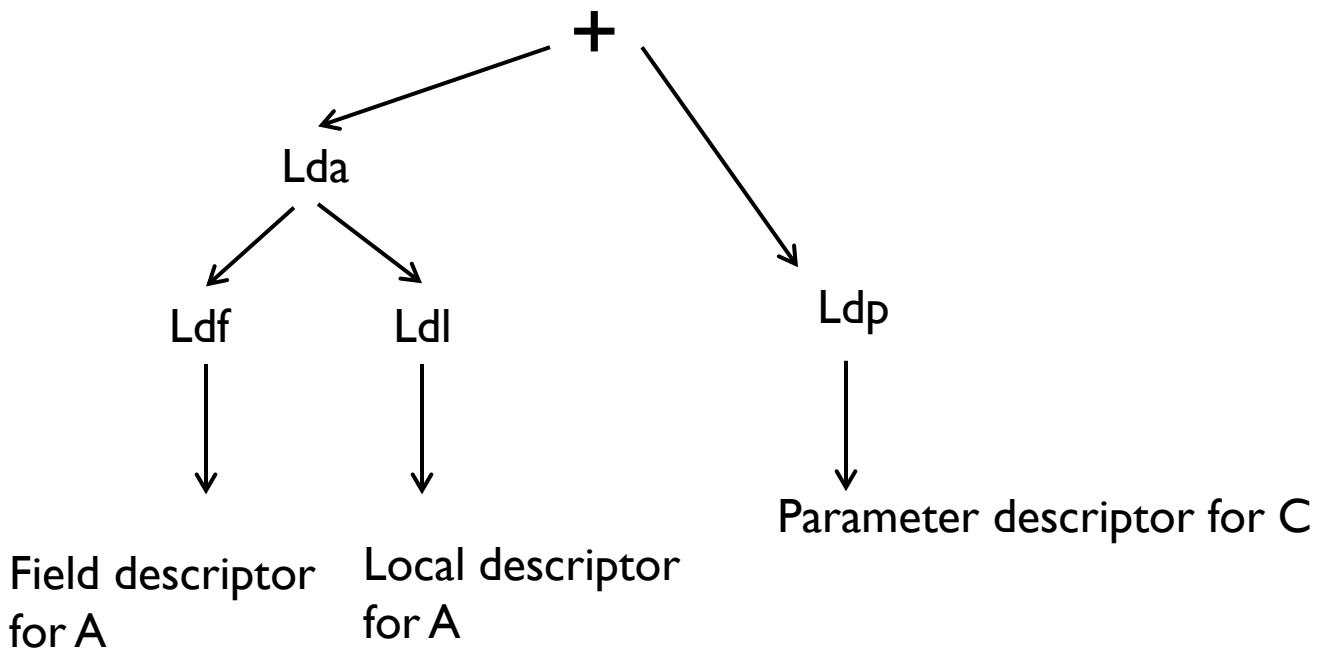
Example 1 (expression)

A*A+C*C



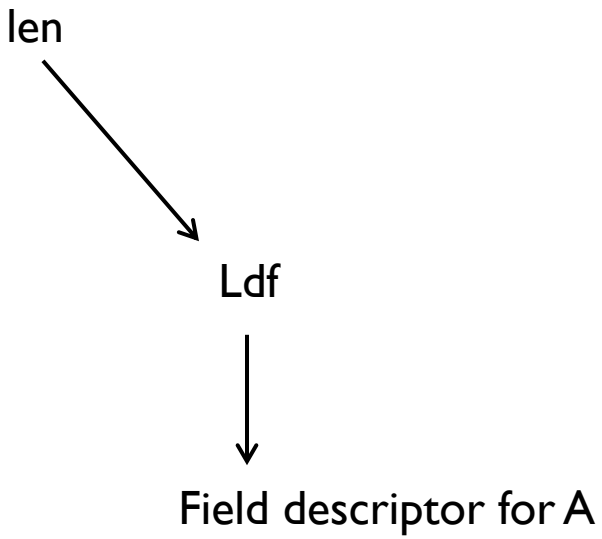
Example 2 (handling array)

A[i]+C



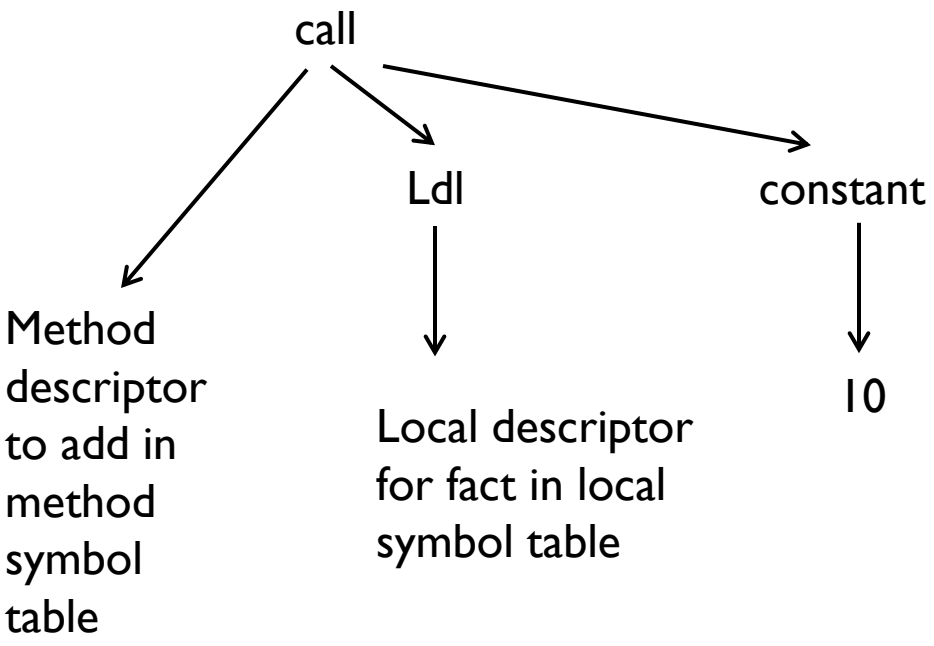
Example 3 (array length)

A.length



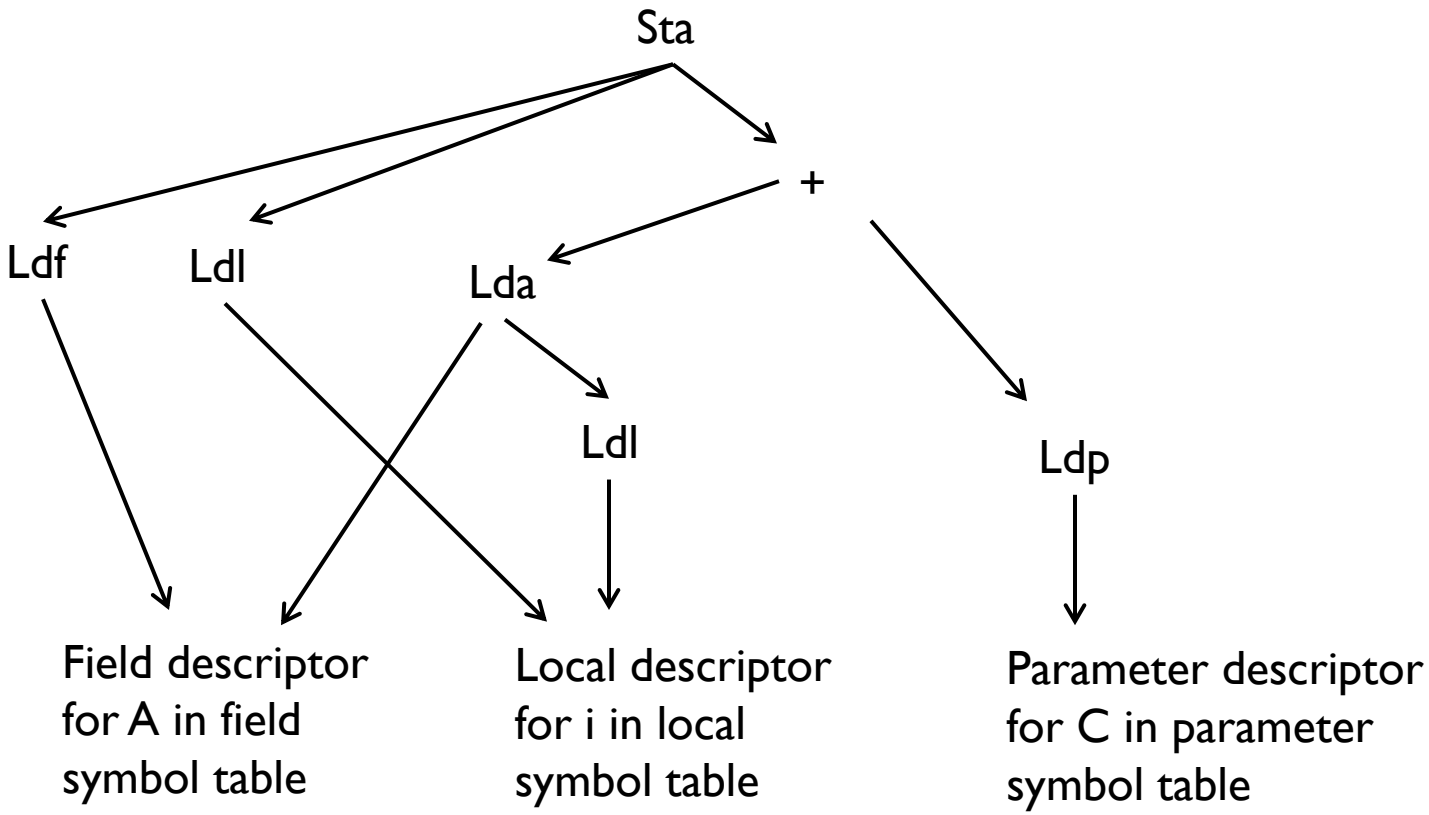
Exmaple 4 (function call)

fact(10)



Example 5

A[i]=A[i]+C



Representing control flow

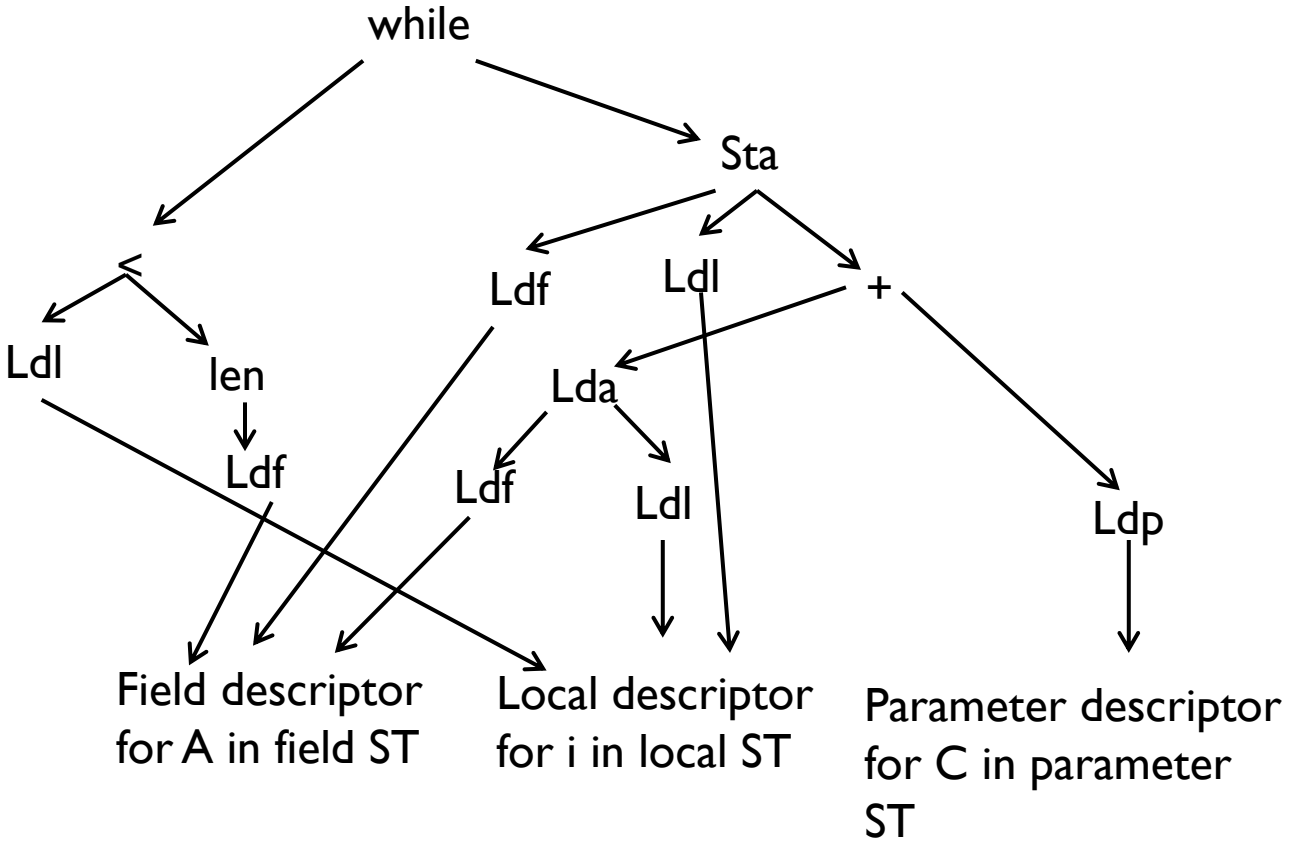
- Statement node
  - Sequence node – first statement, next statement
  - If node
    - Expression tree for condition
    - then statement node, else statement node
  - While node
    - Expression tree for condition
    - Statement node for loop body
  - Return node
    - Expression tree for return value

Abstract syntax tree to intermediate representation (AST to IR)

-

while(i<A.length)

A[i]=A[i]+C



AST to IR

- Recursively traverse AST
- Build up bottom up representation
  - Look up variable identifiers in ST
  - Build load nodes to access variables
  - Build expressions out of load nodes and operator nodes
  - Build store nodes for assignment statements
  - Combine store nodes with control flow nodes

Homework

- Construct the abstract syntax tree (AST) for the following code. Recursively traverse the AST to generate high level intermediate representation (IR).

```
while(i<A.length)
  A[i]=A[i]+C
```

Take home

- High level IR is intuitive to support future compilation tasks
- Representing program data in ST and hierarchically
- Representing computation in expression trees, load store nodes, and control flow structure
- Traverse AST to build IR