Addis Ababa University

College of Natural and Computational Sciences

Department of Computer Science

Compiler and Complexity Module

Part I: Automata and Complexity Theory

Part II: Compiler Design

May 2024

Addis Ababa,

Ethiopia

Compiler Design

Objective of the Course

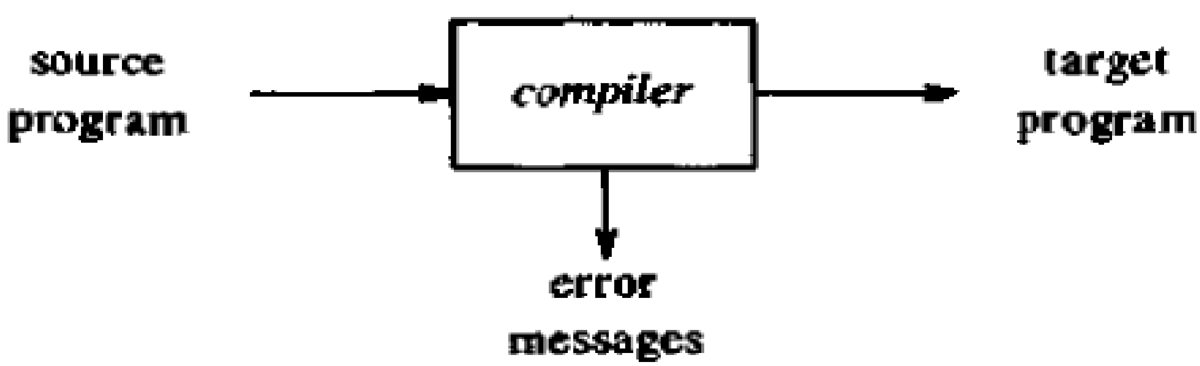
- To learn basic techniques used in compiler construction such as lexical analysis, top-down and bottom-up parsing, context-sensitive analysis, and intermediate code generation.
- To learn basic data structures used in compiler construction such as abstract syntax trees, symbol tables, three-address code, and stack machines.
- To learn software tools used in compiler construction such as lexical analyzer generators, and parser generators.

Chapter One:

Introduction to Compiling

What is Compiler

- a program that reads a program written in one language and translates it into an equivalent program in another language.



Compiler vs Interpreter

- **Compiler:** convert human readable instructions to computer readable instructions one time.
- **Interpreter:** converts human instructions to machine instructions each time the program is run.

Applications of compiler technology

- Parsers for HTML in web browser
- Machine code generation for high level languages
- Software testing

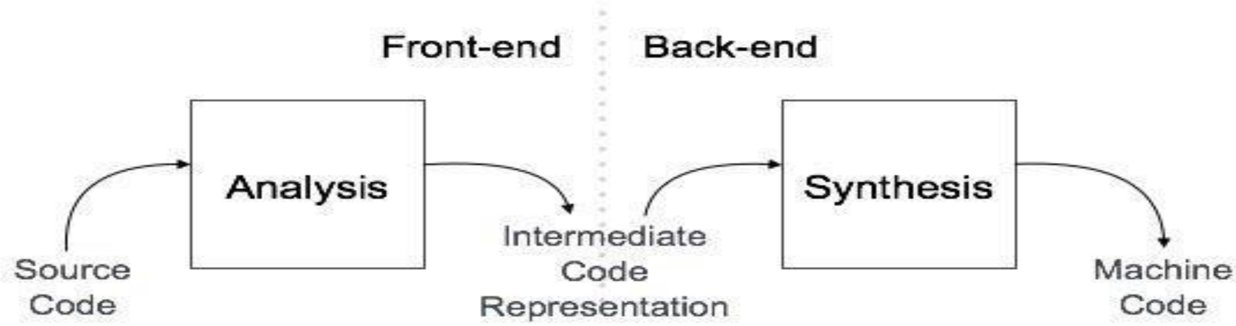
- Program optimization
- Malicious code detection
- Design of new computer architectures

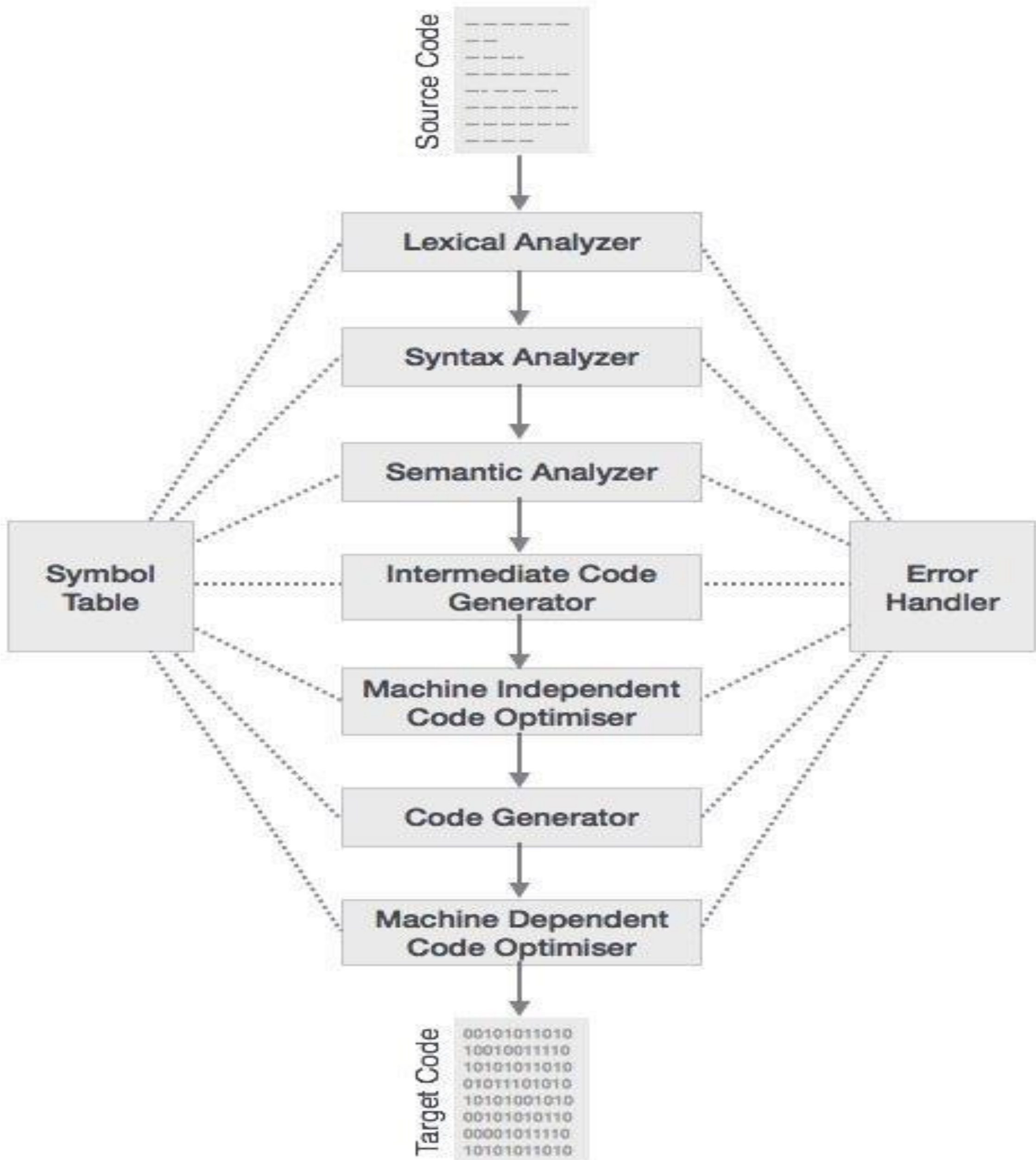
Cousins of the Compiler

- Preprocessor:
  - produces input for compiler
  - file inclusion, language extension, etc.
- Assembler
  - assembly language into machine code
  - output of an assembler is called an object file
- Linker
  - links and merges various object files to make an executable file.
  - determine the memory location where these codes will be loaded
- Loader
  - loading executable files into memory and execute them.
  - It calculates the size of a program (instructions and data) and creates memory space for it.
  - It initializes various registers to initiate execution.
- Cross-Compiler
  - compiler that runs on platform (A) and generates executable code for another platform (B).
- Source-to-source Compiler
  - compiler that translates source code of one programming language to another

Phases of a Compiler

- **Analysis**
  - Machine Independent/Language Dependent
- **Synthesis**
  - Machine Dependent/Language independent

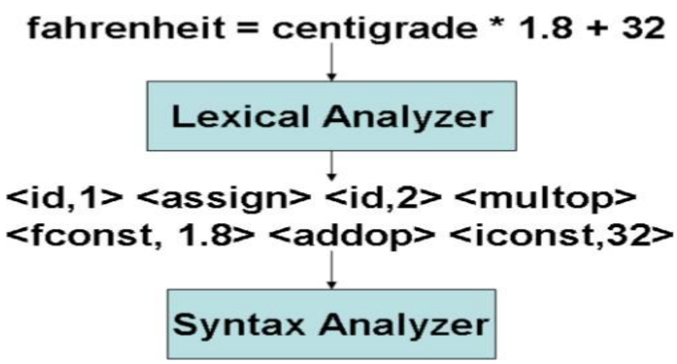

Front-end
Back-end
Analysis
Synthesis
Source Code
Intermediate Code Representation
Machine Code


Source Code
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Machine Independent Code Optimiser
Code Generator
Machine Dependent Code Optimiser
Symbol Table
Error Handler
Target Code
00101011010
10010011110
10101011010
01011101010
10101001010
00101010110
00001011110
10101011010

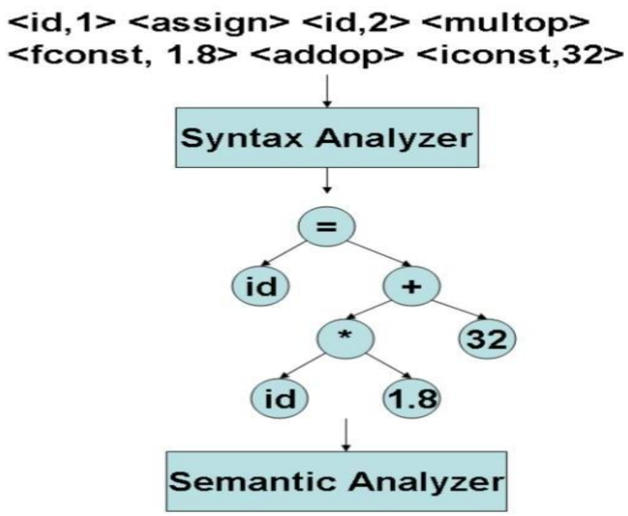
Analysis of the Source Program

1. **Lexical / Linear Analysis (scanning)**
   - Scans the source code as a stream of characters
   - Represent lexemes in the form of tokens as:

     <token-name, attribute-value>
   - Token
     - smallest meaningful element that a compiler understands.
   - Eg.
     - Identifiers, Keywords, Literals, Operators and Special symbols.
   - Blanks, new lines, comments will be removed from the source program.

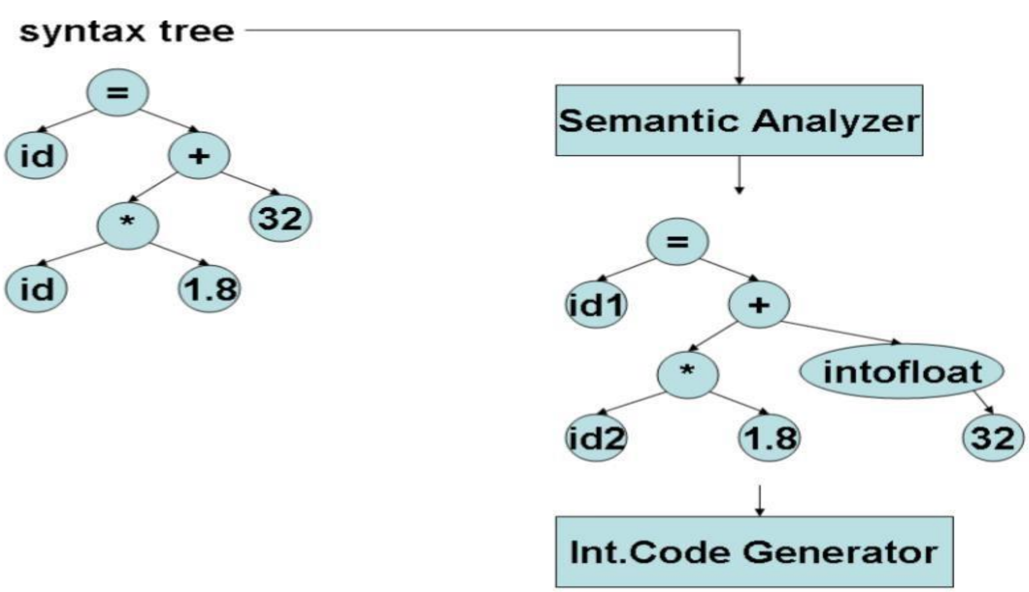


2. **Syntax / Hierarchical Analysis – Parsing**
   - Tokens are grouped hierarchically into nested collections with collective meaning.
   - The result is generally a parse tree.
   - expressions, statements, declarations etc... are identified by using the results of lexical analysis.
   - Most syntactic errors in the source program are caught in this phase.
   - Syntactic rules of the source language are given via a Grammar.

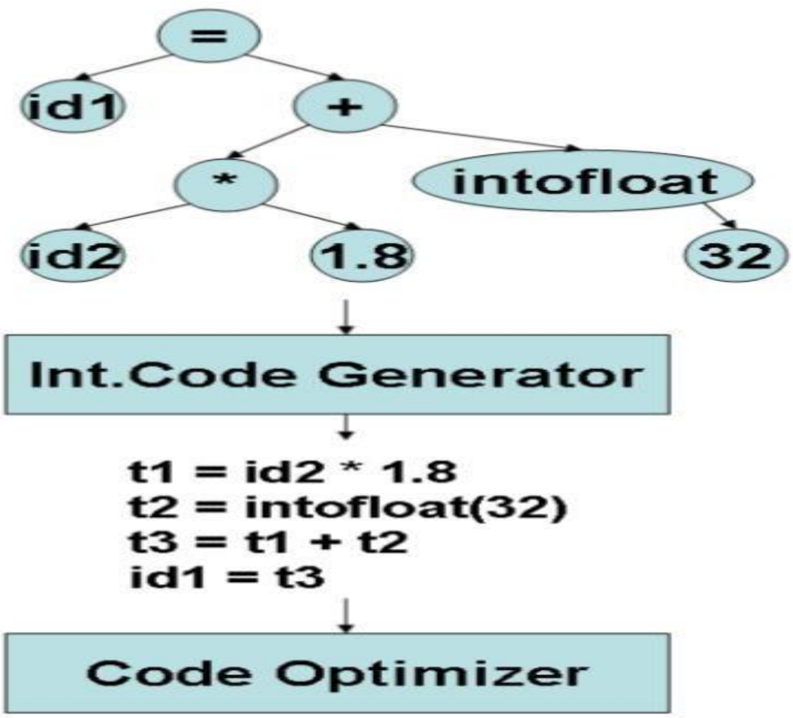


3. **Semantic Analysis**

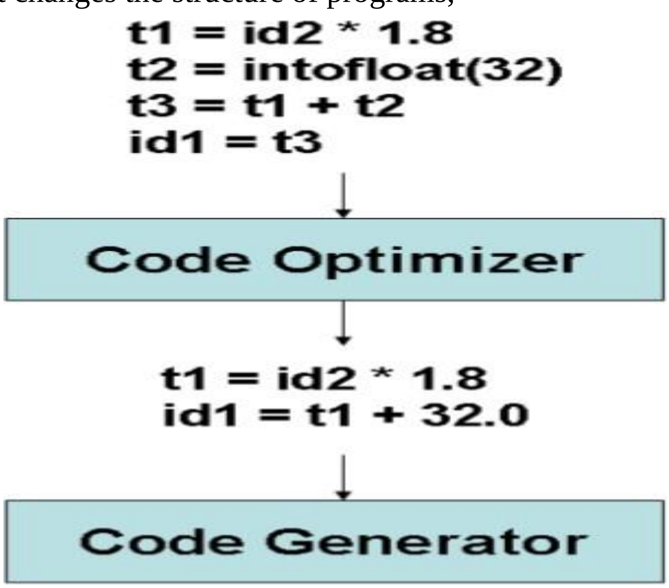
- Certain checks are performed to make sure that the components of the program fit together meaningfully.
- Unlike parsing, this phase checks for semantic errors in the source program (e.g. type mismatch)
  - Type checking of various programming language constructs is one of the most important tasks.
- Stores type information in the symbol table or the syntax tree.
  - Types of variables, function parameters, array dimensions, etc.



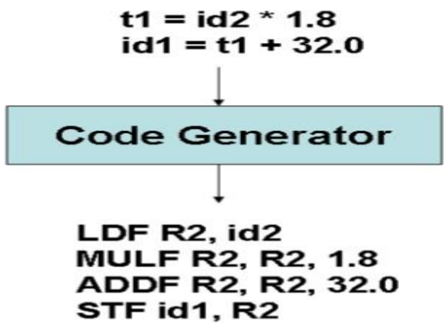
4. **Intermediate Code Generation**
   Easy to produce and easy to translate to machine code



5. **Code Optimization**
   Changes the IC by removing such inefficiencies
   Improve the code
   a. Improvement may be time, space, or power consumption.
   It changes the structure of programs,



6. **Code Generation**
   Converts intermediate code to machine code.
   Must handle all aspects of machine architecture
   Storage allocation decisions are made
   a. Register allocation and assignment



Chapter 2:

Lexical Analysis

What is Lexical Analysis

- The first phase of a compiler
- The input is a high level language program
- The output is a sequence of tokens
- Strips off blanks, tabs, newlines, and comments from the source program
- Keeps track of line numbers

Tokens, Patterns, and Lexemes

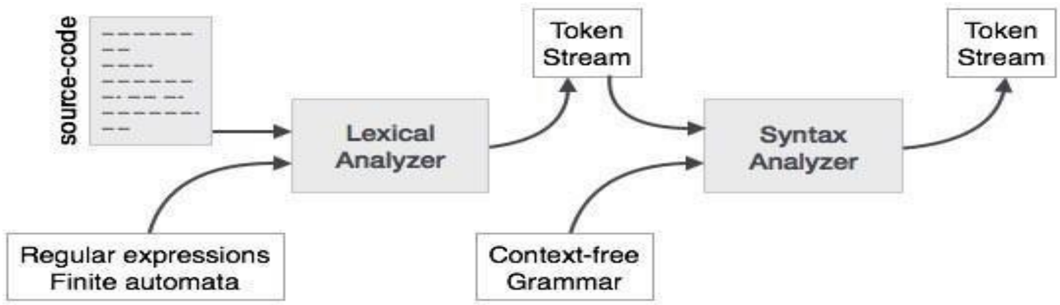
- Token
  - A string of characters which logically belong together
  - Classes of similar lexemes
    - identifier, keywords, constants etc.
- Pattern
  - A rule which describes a token
- Lexeme
  - The sequence of characters matched by a pattern to form the token

- **Classes of Tokens**
  - **Identifiers:** names chosen by the programmer
  - **Keywords:** names already in the programming language
  - **Separators:** punctuation characters
  - **Operators:** symbols that operate on arguments and produce results
  - **Literals:** numeric, textual literals

Chapter 3

Syntax Analysis

- Every language has rules for syntactic structure of well formed programs.
- Takes streams of tokens from lexical analyzer and produce a parse tree.



Grammars

- Every programming language has grammar rules
- Parsers or syntax analyzers are generated for a particular grammar
- CFG are used for syntax specification of programming languages

Context Free Grammar (CFG)

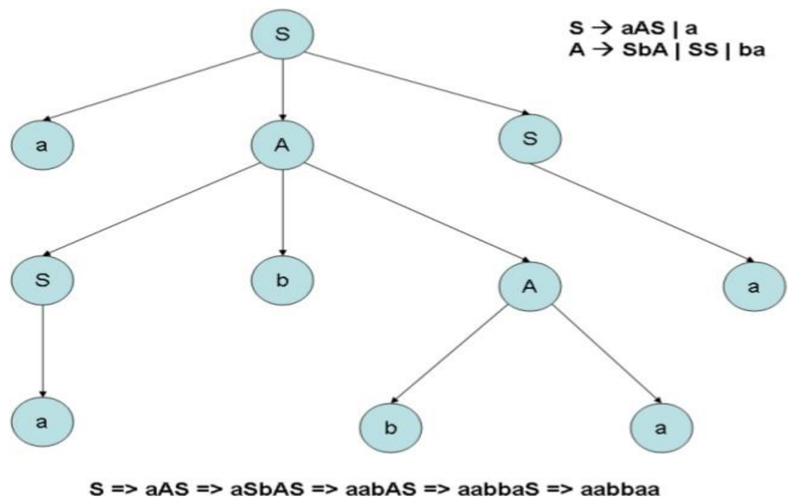
- Is denoted as G = (N, T , P, S)
- N : finite set of non-terminals
- T : finite set of terminals
    - $S \in N$: The start symbol
    - P : Finite set of productions, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$

Derivations

- Derivation of terminal string from non-terminal
- A production is applied at each step in derivation
- the productions $E \rightarrow E + E$, $E \rightarrow id$, and $E \rightarrow id$, are applied at steps 1,2, and, 3 respectively.
- read as S derives id + id.

Derivation Trees

- Derivations can be displayed as trees
- Internal nodes of the tree are all non-terminals
- Leaves are all terminals
- The yield of a derivation tree is the list of the labels of all the leaves read from left to right.



Leftmost and Rightmost Derivations

- Leftmost Derivation
  - Apply a production only to the leftmost variable at every step
  - S → aAS | a | SS
  - A → SbA | ba
  - S => aAS => aSbAS =>aabAS => aabbaS => aabbaa
- Rightmost Derivation
  - Apply production to the rightmost variable at every step
  - S =>aAS =>aAa=>aSbAa =>aSbbaa =>aabbaa

Parsing

- Process of constructing parse tree for a sentence generated by a given grammar.
- 2 types of parsers
  - Top down parsing (predictive parsers)
    - LL(1)
  - Bottom up parsing (SR parsers)
    - LR(1)

Top Down Parsing

- The parse tree is created top to bottom
- Starts from the start symbol and transform it to the input

Bottom Up Parsing

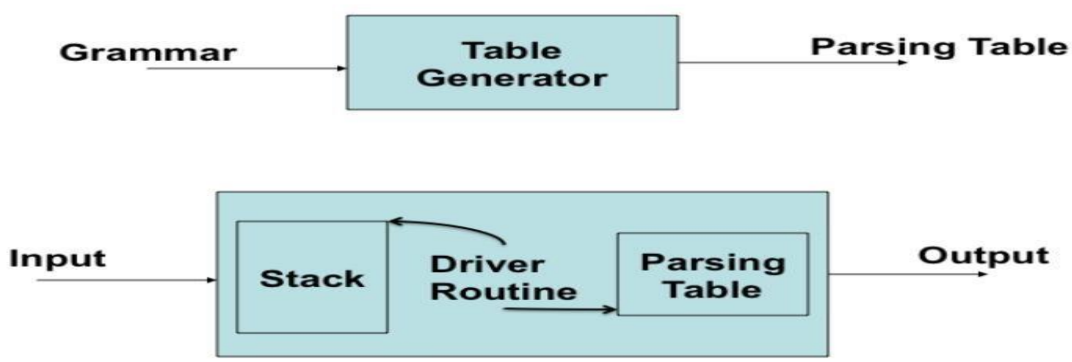
- Starts with the input symbols and tries to construct the parse tree up to the start symbol.
- One way of reducing a sentence is to follow the right most derivation in reverse

LL(1) Grammar

- L – left to right
- L – left most derivation
- 1 – number of look ahead
- First( ) and Follow( )
  - the first terminal in a string and the terminal that follows a variable respectively.

LR Parsing

- LR(k) - Left to right scanning with Rightmost derivation in reverse, k being the number of lookahead tokens.



LR Parser Generator

Types of LR Parsers

- LR (0) , SLR (1) , LALR (1) , CLR (1)

LL	LR
Leftmost derivation	Rightmost derivation in reverse
Starts with root non-terminal on stack	Ends with root non-terminal on the stack
Builds the parse tree top-down	Builds the parse tree bottom-up
Expands the non-terminals	Reduces the non-terminals
Ends when the stack is empty	Starts with an empty stack

Chapter 4

Semantic Analysis

Syntax Directed Translation

- Attaching actions to the grammar rules(productions).
- Actions are executed during the compilation
  - Not during the generation of the compiler
- Actions are executed according to the parsing mechanism.

Syntax Directed Definitions

- Is a generalization of a context free grammar
- Is a CFG with attributes and rules
- Attributes are associated with grammar symbols and rules with productions
- Attributes may be:
  - Numbers
  - Types
  - Strings etc

Syntax Directed Definition- Example

Production	Semantic Rules
L → E return	print(E.val)
E → E1 + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T1 * F	T.val = T1.val * F.val
T → F	T.val = F.val
T → (E)	F.val = E.val
F → digit	F.val = digit.lexval

Functions for Syntax Tree Nodes

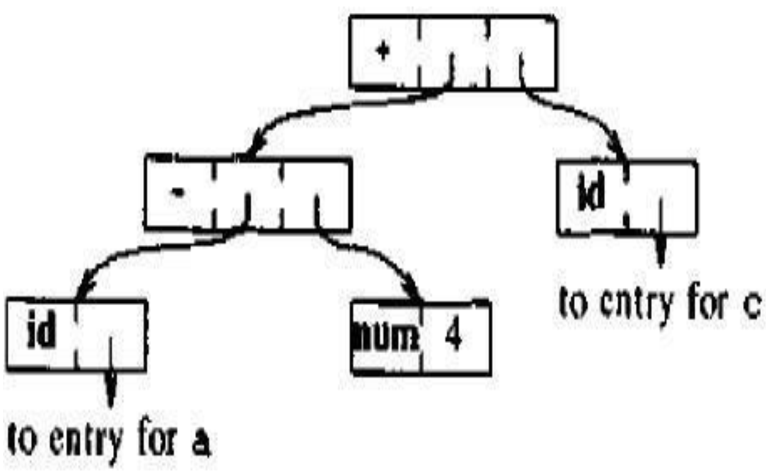
- **mknode (op, left, right)**
  - Creates an operator node with label op &

- Two fields containing pointers to left and right

- **mkleaf(id, entry)**
  - Creates an identifier node with label id &
  - A field containing entry, ptr to symbol table entry for the identifier
- **mkleaf(num, val)**
  - Create a number node with label num &
  - A field containing val, the value for the number

Syntax tree for expression a-4+c

- P1=mkleaf(id,entrya);
- P2=mkleaf(num, 4);
- P3=mknode('-',p1,p2);
- P4=mkleaf(id,entryc);
- P5=mknode('+',p3,p4);



Chapter 5
Type Checking

What are Types ?

- **Types:**
  - Describe the values computed during the execution of the program

- **Type Errors:**
    - Improper or inconsistent operations during program execution
- **Type-safety:**
    - Absence of type errors

Type Checking

- Semantic checks to enforce the type safety of the program
- Semantic Checks
    - Static – done during compilation
    - Dynamic – done during run-time
- Examples
    - Unary and binary operators
    - Number and type of arguments
    - Return statement with return type
    - Compatible assignment

Static Checking

- The compiler must check the semantic conventions of the source language
- Static Checking: ensures that certain kind of errors are detected and reported
- Example
- Type Checks: incompatible operands
- Flow Control Check
- Uniqueness Check
- Name Related Check

Type Checking of Expressions

E → literal { E.type = char }

E → num { E.type = int }

E → id { E.type = lookup(id.entry) }

E→E1 mod E2 { E.type=if E1.type=int and E2.type= int then int

else type_error }

E→E1[E2] { E.type=if E2.type=int and E1.type=array(s,t) then t else type_error }

Type Checking of Statements

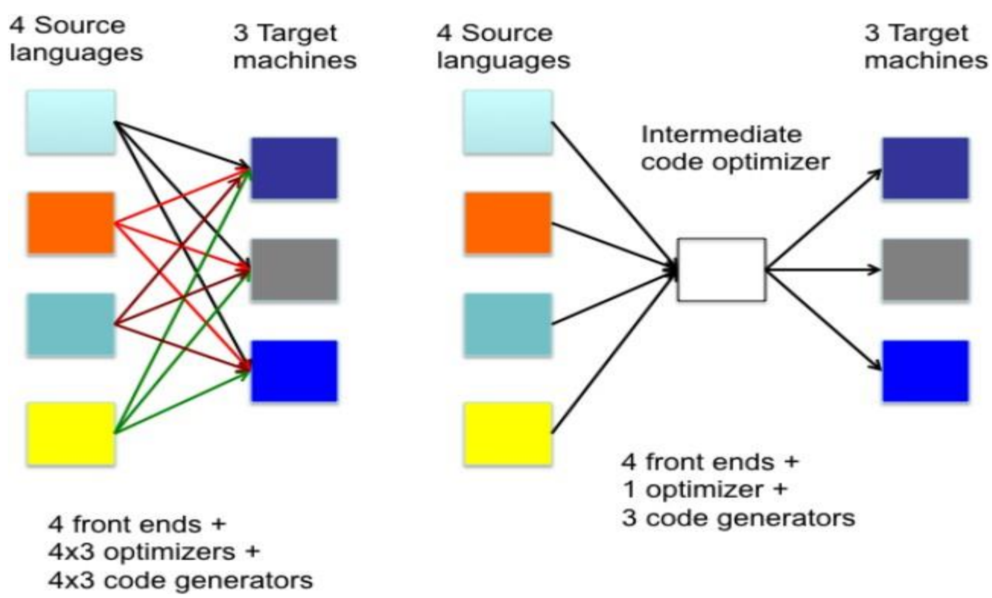
S→id=E { S.type = if id.type=E.type then void else type_error }

S→if E then S1 { S.type = if E.type=Boolean then S1.type else type_error }

S→while E do S1 { S.type = if E.type = Boolean then S1.type else type_error }

Chapter Six

Intermediate Code Generation



Three Address Code

- Is a sequence of statements of the form
  - X = Y op Z
  - X,Y and Z are names, constants or compiler generated temporaries
  - Op is operator (arithmetic, logical)
- Example:
  - a = b + c , x = -y , if a > b goto L1
- LHS is the target
- RHS has at most two sources and one operator

Three Address Code

- Is a generic form and can be implemented as:
  - Quadruples
  - Triples
  - Indirect Triples
  - Tree
  - DAG
- Example: a = b + c * d , a + b * c - d / (b * c) ?
- t1 = c * d
- t2 = b + t1
- a = t2

Three Address Code

- Quadruples:
  - Each instruction is divided into four fields
  - Operator, arg1, arg2, and result
- Triples:
  - Has three fields
  - Operator, arg1 and arg2
- DAG and Tree
  - Similar presentation of expression to triples
- Indirect Triples
  - Uses pointers instead of position to store results

Implementations of 3-Address Code

3-address code

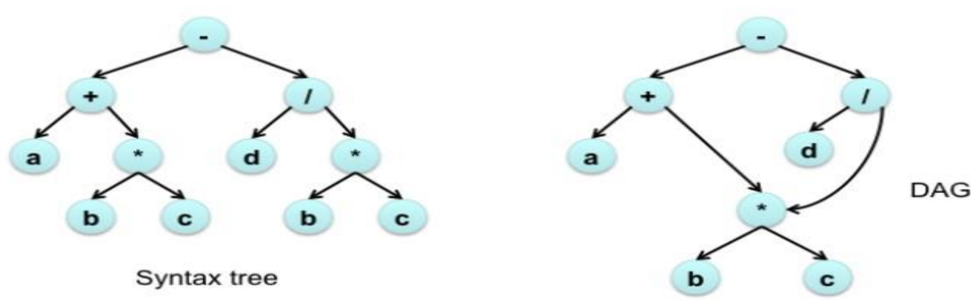
1. $t_1 = b*c$
2. $t_2 = a + t_1$
3. $t_3 = b*c$
4. $t_4 = d/t_3$
5. $t_5 = t_2-t_4$

Quadruples

op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Triples

	op	arg_1	arg_2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)



C-Program

```
int a[10], b[10], dot_prod, i;
dot_prod = 0;
for (i=0; i<10; i++) dot_prod += a[i]*b[i];
```

Intermediate code

```
    dot_prod = 0;            |    T6 = T4[T5]
    i = 0;                   |    T7 = T3*T6
L1: if(i >= 10)goto L2       |    T8 = dot_prod+T7
    T1 = addr(a)             |    dot_prod = T8
    T2 = i*4                 |    T9 = i+1
    T3 = T1[T2]              |    i = T9
    T4 = addr(b)             |    goto L1
    T5 = i*4                 |L2:
```

Declarations

- Involves allocation of space in memory &
- Entry of type and name in symbol table
- Off set variable (Offset=0) is used to denote the base address

int a; float b;

Allocation process: { offset = 0 }

int a;

id.type = int

id.width = 2

offset = offset + id.width { offset = 2 }

float b;

id.type=float

id.width=4

offset = offset +id.width { offset = 6 }

Chapter 8
Introduction to Code Optimization

Goals of Code Optimization

- Remove redundant code without changing the meaning of program
- Executes faster
- Efficient memory usage
- Better performance

Techniques

- Common sub-expression elimination
  - Repeated appearance computed previously
- Strength reduction
  - Replacement of expensive expressions with simple ones
- Code movement
  - Moving a block of code outside a loop
- Dead code elimination
  - Eliminated code statements that are either never executed or unreachable

Register Allocation

- Registers hold values
- Example
  - a = c + d
  - e = a + b
  - f = e – 1
- With the assumption that a and e die after use
- Temporary a can be reused after e=a+b, same wz a
- Can allocate a,e and f all to one register(r1)
  - r1 = r2 + r3
  - r1 = r1 + r4
  - r1 = r1 – 1

Peephole Optimization

- Transforming to optimal sequence of instructions

Common Techniques:

- Elimination of redundant loads and stores
  - Eg.
  - r2 = r1 + 5
  - I = r2
  - r3 = I
  - r4 = r3 * 3
- Constant folding
  - Eg.
  - R2 = 3 * 2
- Constant Propagation
  - Eg.
  - r1 = 3
  - r2 = r1 * 2
- Copy Propagation
  - Eg.
  - r2 = r1

- r3 = r1 + r2
- r2 = 5;

- Elimination of useless instructions
  - Eg.
  - r1 = r1 + 0   r1 = r1 * 1