CS49K Programming Languages

Chapter 4 Semantic Analysis
Additional Material: GNU Compiler

LECTURE 6A: THE STRUCTURE OF A COMPILER (GNU COMPILER)

DR. ERIC CHOU

IEEE SENIOR MEMBER

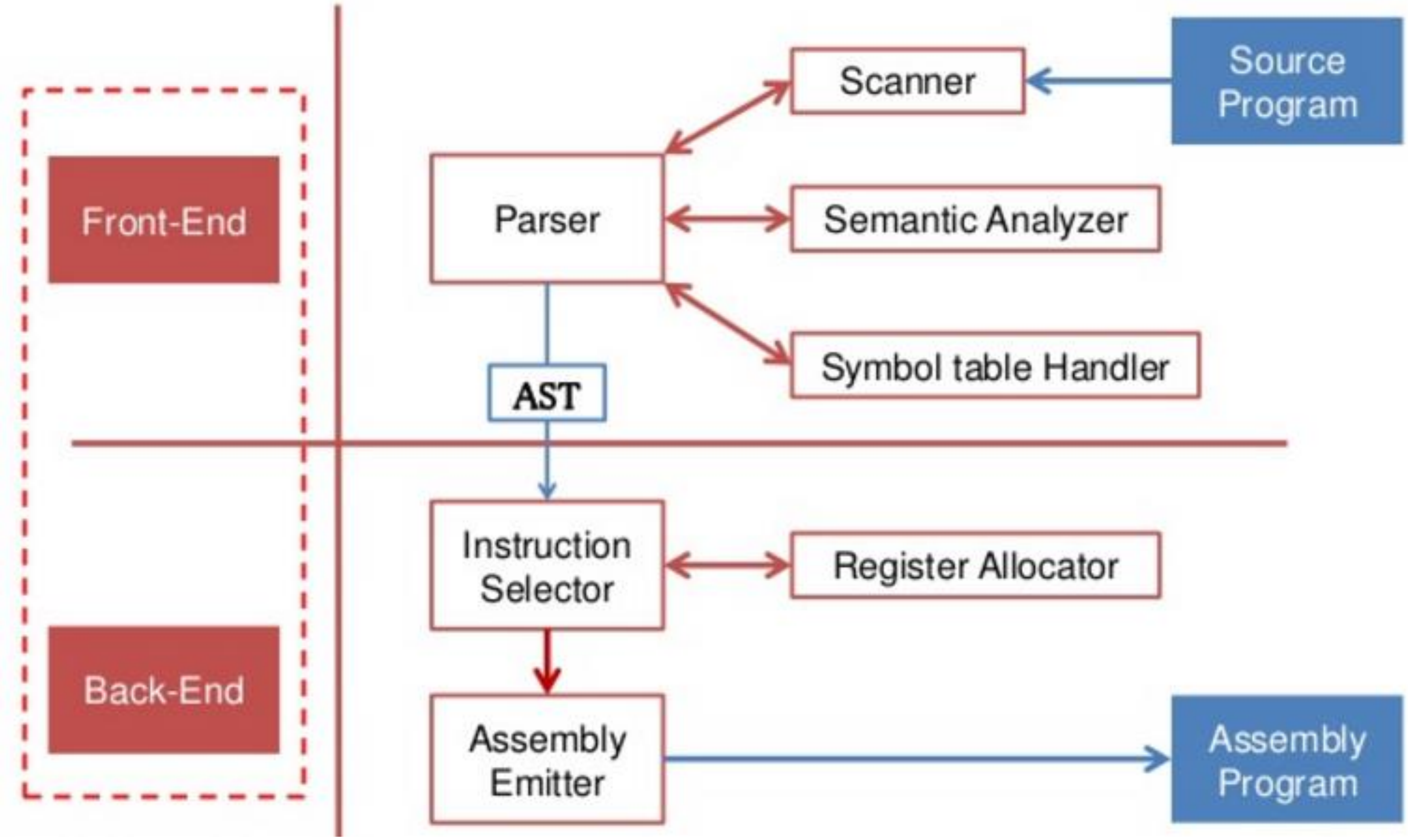
Objectives

- Structure of a Compiler
- GNU Compiler Collection
- The role of Semantic Analysis

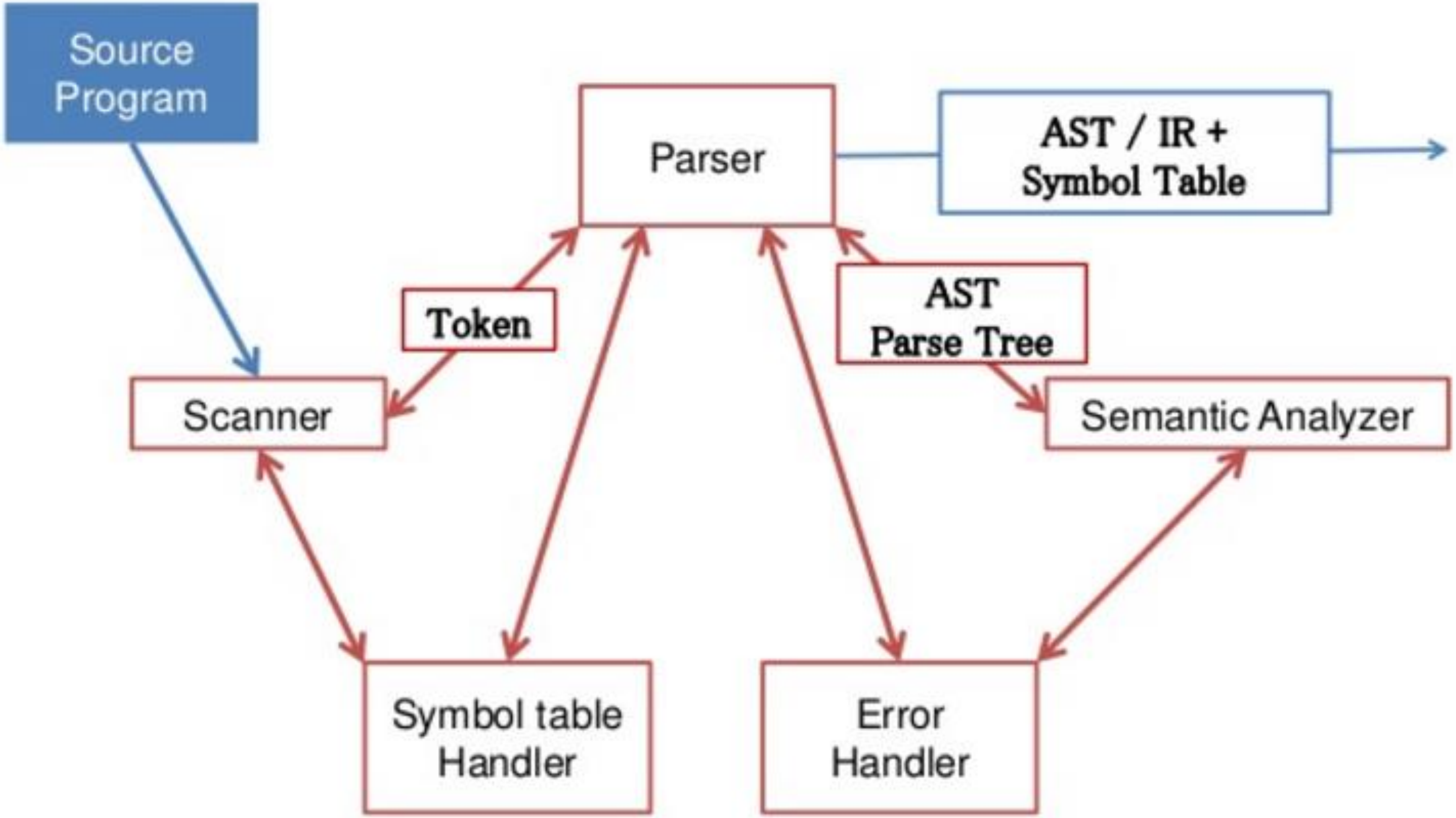
Overview of Semantic Analysis in Compilation Flow

SECTION 1

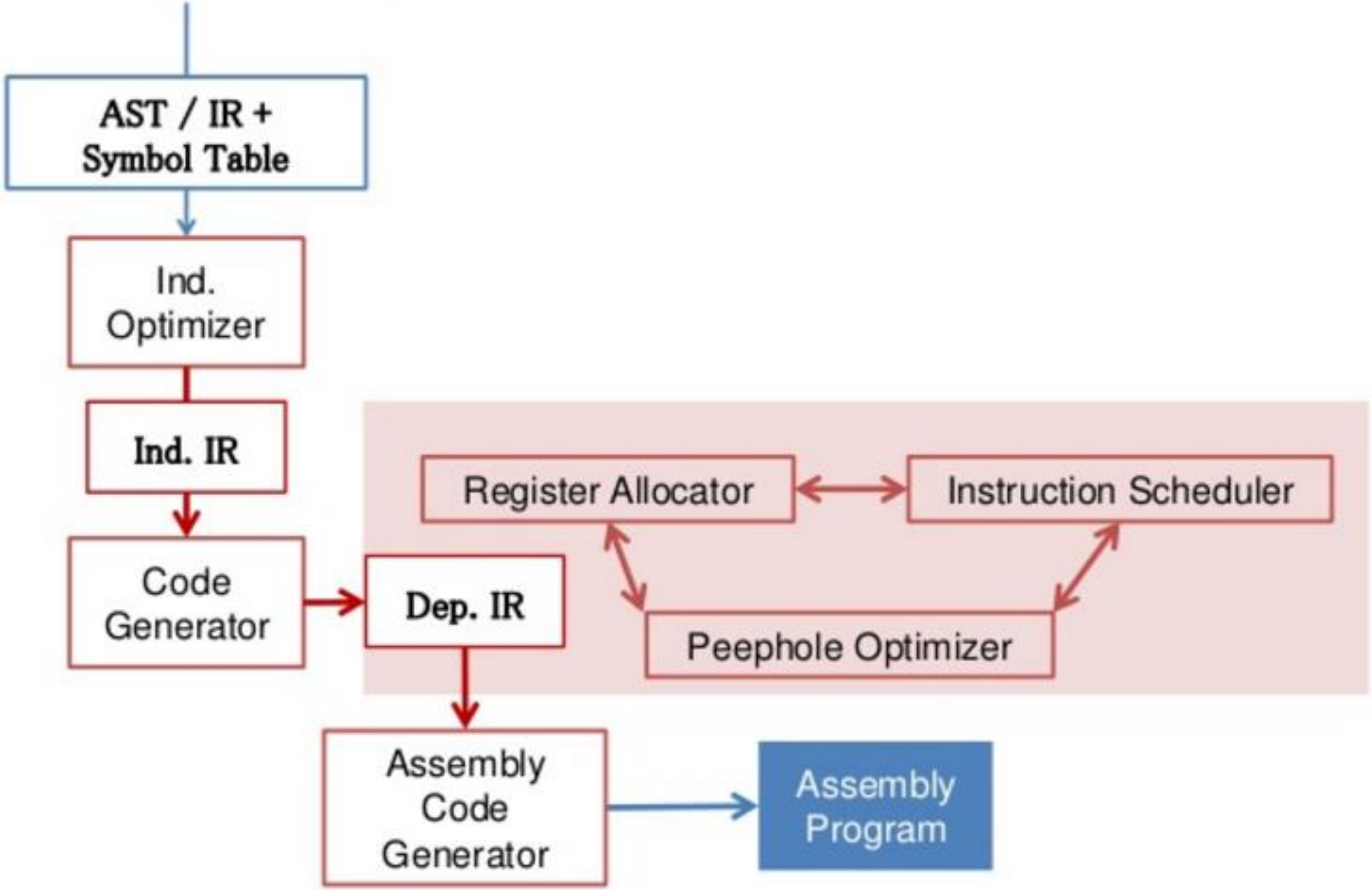
Structure of Compiler



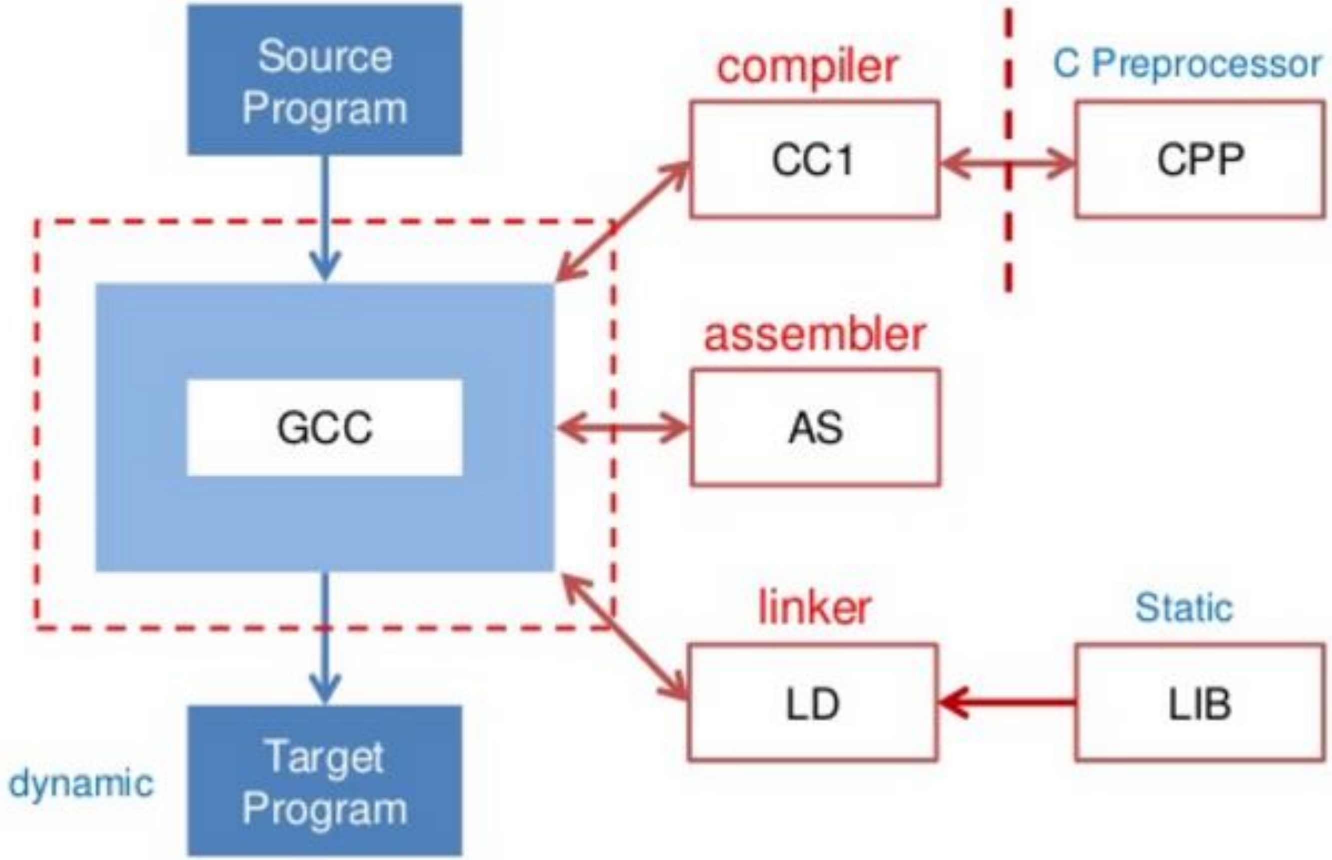
Typical Front-End



Typical Back-End



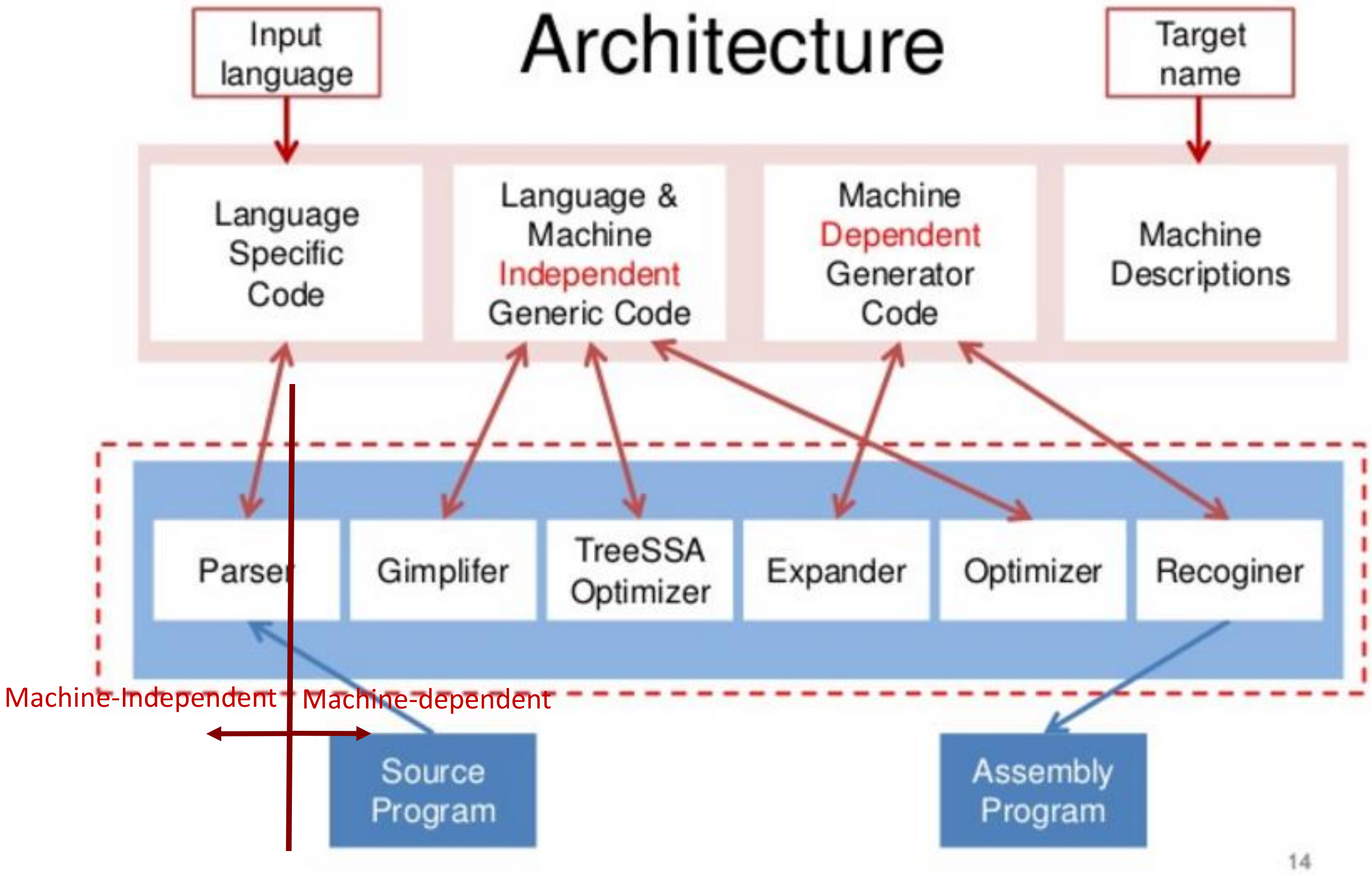
GCC compiler



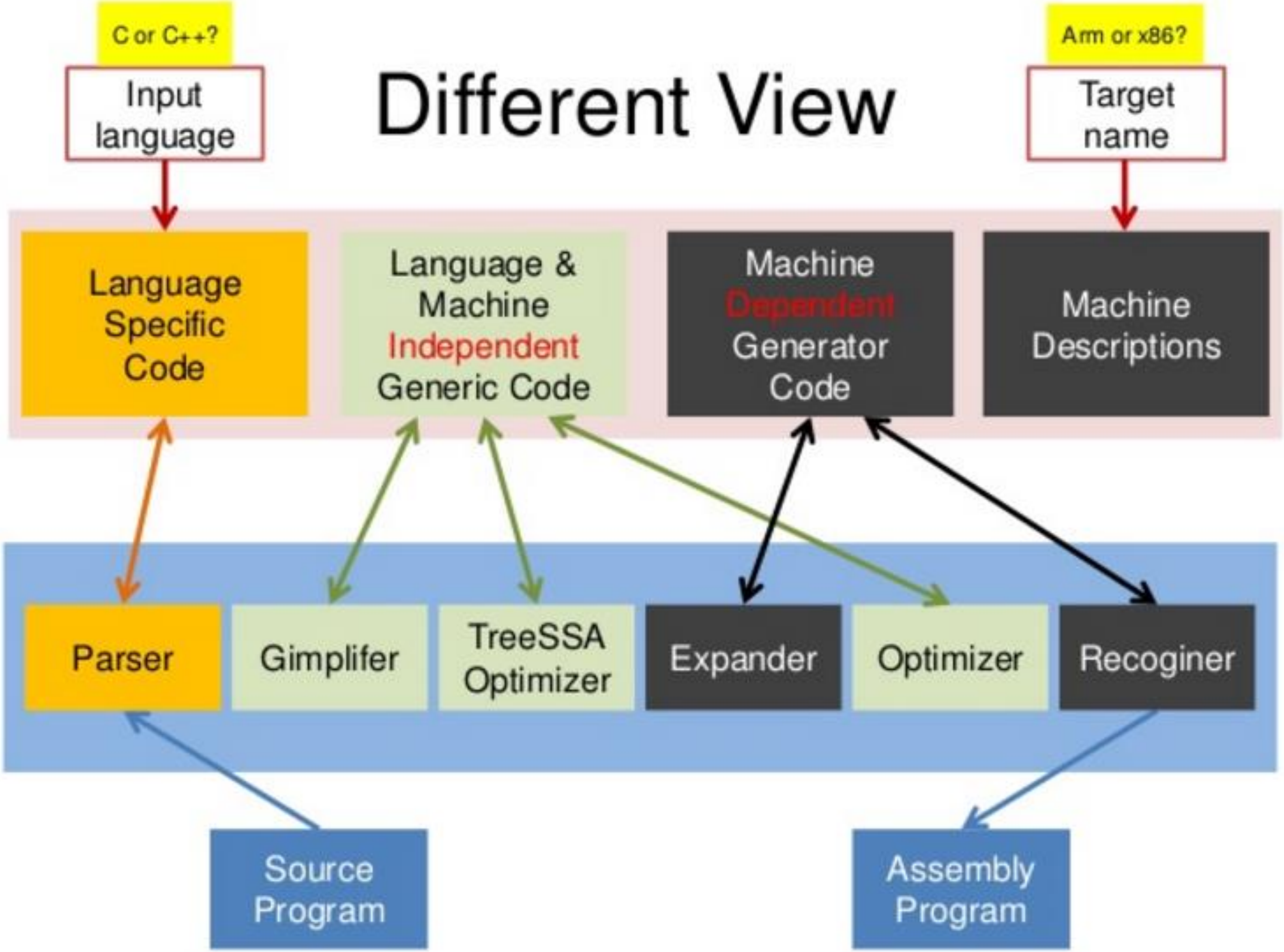
GCC is a collection that invokes compiler, assembler and linker…

GCC Compiler Architecture

GNU Compiler Collection

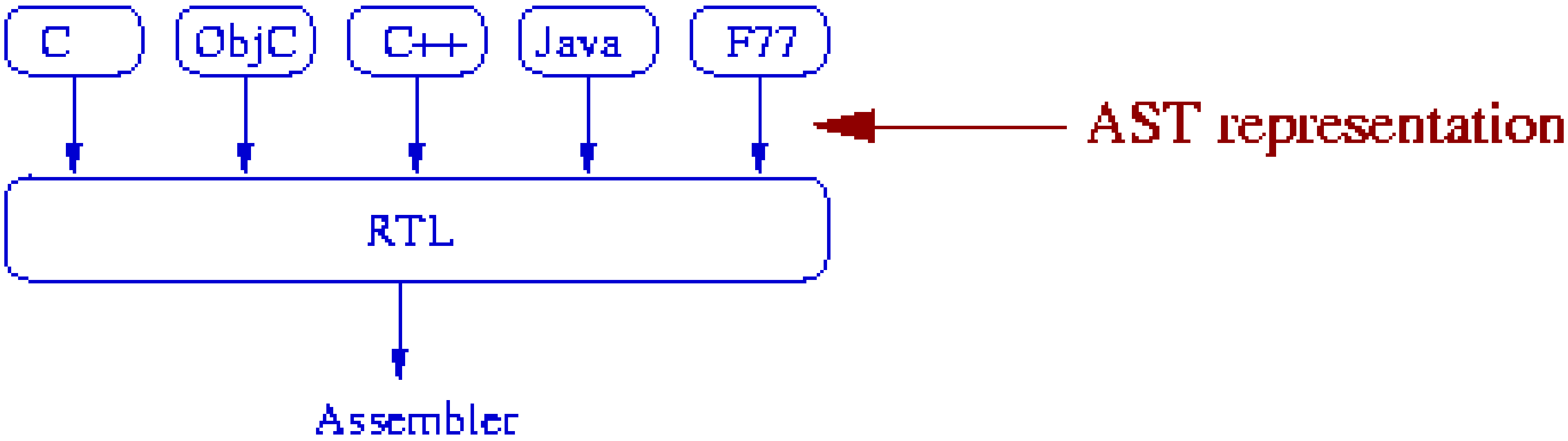


Different View



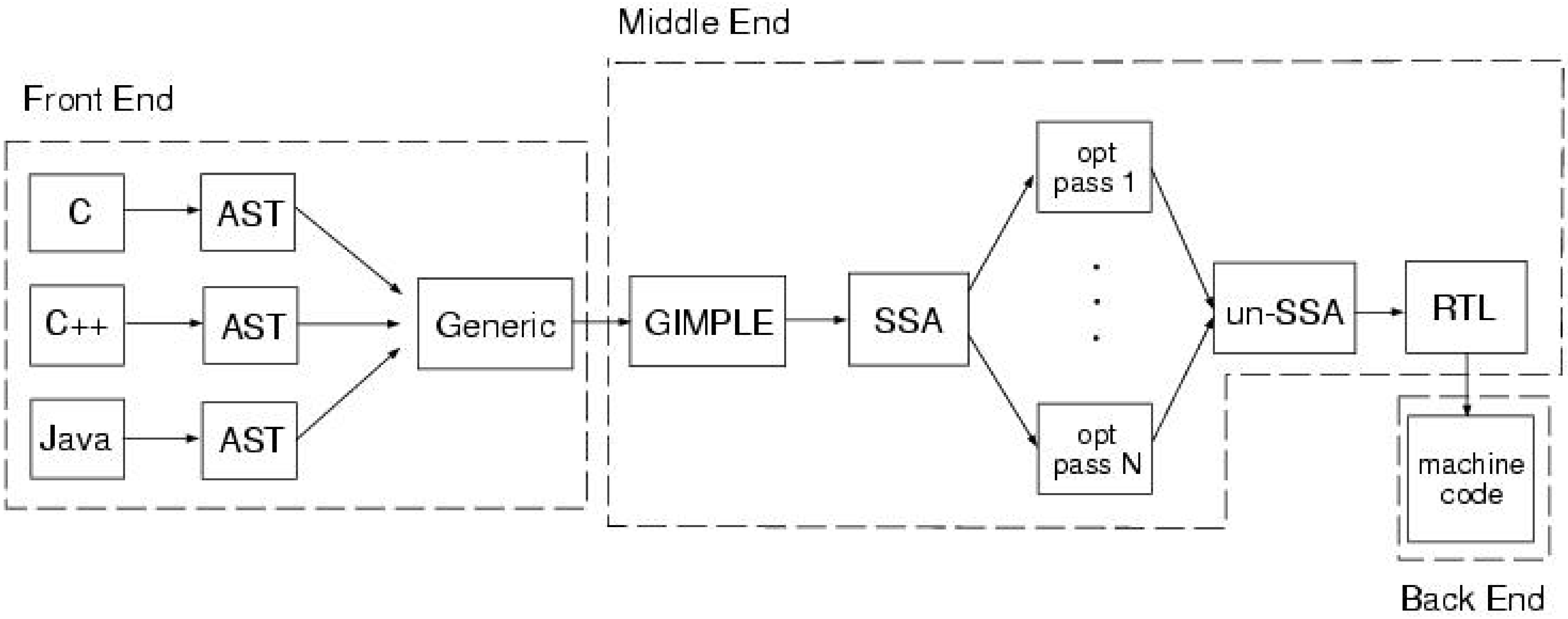
Note:

- The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project supporting various programming languages.
- GCC is a key component of the GNU toolchain and the standard compiler for most Unix-like Operating Systems.
- GCC has played an important role in the growth of free software, as both a tool and an example.
- Originally named the GNU **C** Compiler, when it only handled the **C** programming language, GCC 1.0 was released in 1987.
- It was extended to compile **C++** in December of that year.
- Front ends were later developed for **Objective-C, Objective-C++, Fortran, Java, Ada,** and **Go** among others.
- Version 4.5 of the OpenMP specification is now supported in the C and C++ compilers.
- An improved implementation of the OpenACC 2.0a specification is also supported.
- The current version supports gnu++14, a superset of C++14 and gnu11, a superset of C11, with strict standard support also available.
- It also provides experimental support for C++17 and later.


C
ObjC
C++
Java
F77
RTL
Assembler
AST representation

Data Flow View

C, C++, Objective-C, Objective-C++, Fortran, Java, Ada, and Go

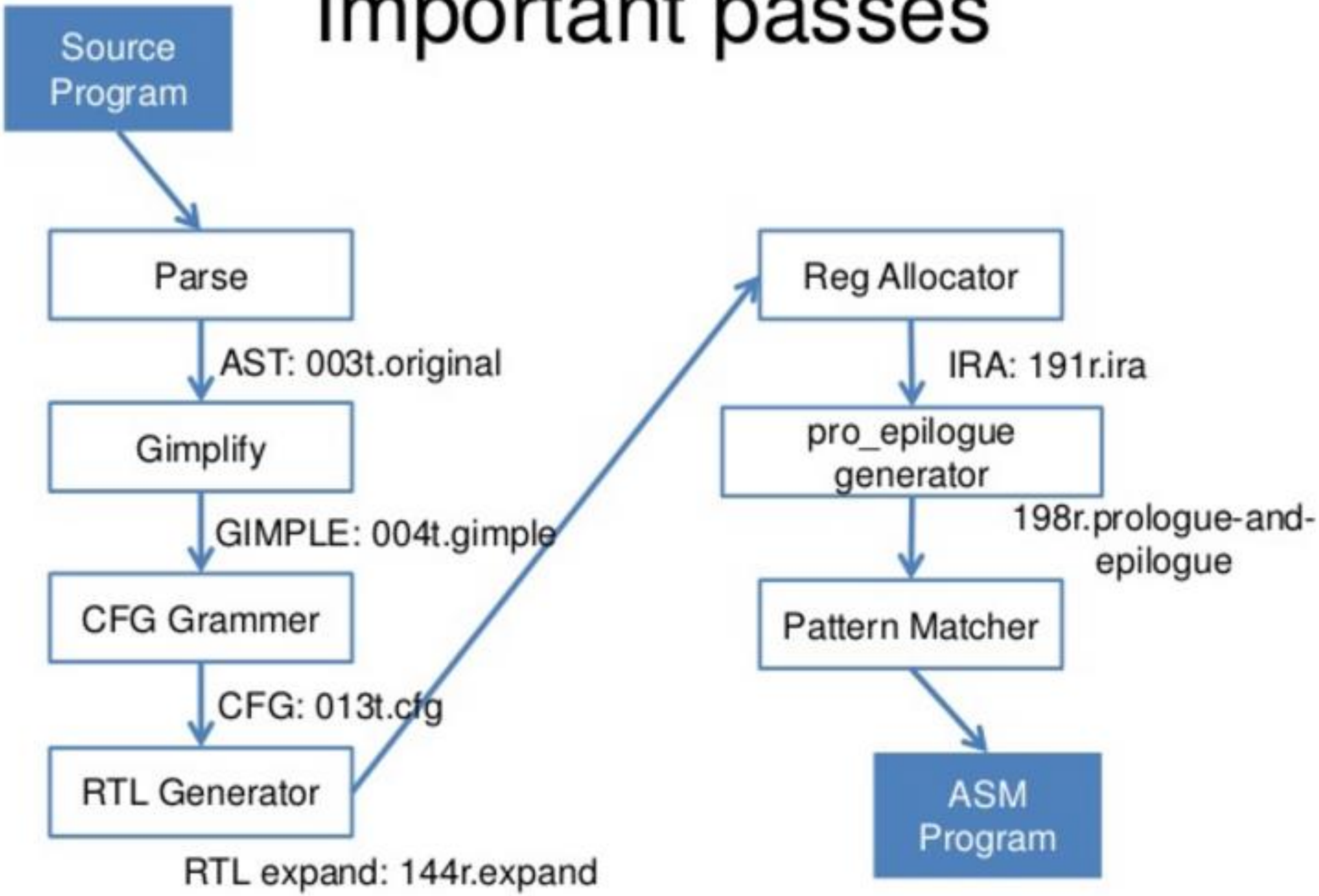


AST : Abstract Syntax Trees are produced by each front-end as an intermediate representation. They are then translated to RTL after some analyses and optimizations.
GENERIC: The purpose of GENERIC is simply to provide a language-independent way of representing an entire function in trees.
GIMPLE: GIMPLE is a much more restrictive representation than abstract syntax trees (AST). GIMPLE is a three-address representation derived from GENERIC by breaking down GENERIC expressions into tuples of no more than 3 operands.
SSA: Most of the tree optimizers rely on the data flow information provided by the Static Single Assignment (SSA) form. This is a representation of the program where each variable is assigned exactly once. Multiple assignments to the same variable in the source code are rewritten as assignments to subscripted independent variables.
RTL: The last part of the compiler work is done on a low-level intermediate representation called Register Transfer Language. In this language, the instructions to be output are described, pretty much one by one, in an algebraic form that describes what the instruction does.

The Compilation Flow of the GNU Compiler Collection

SECTION 2

Important passes



Examples: AST dumps

1. gcc -fdump-tree-original-raw test.c

test.c

```c
int a;
main()
{
    a = 55;
}
```

```
@1      bind_expr        type: @2         body: @3
@2      void_type        name: @4         algn: 8
@3      modify_expr      type: @5         op 0: @6         op 1: @7
@4      type_decl        name: @8         type: @2
@5      integer_type     name: @9         size: @10        algn: 32
                         prec: 32         sign: signed     min : @11
                         max : @12
@6      var_decl         name: @13        type: @5         srcp: test.c:1
                         size: @10        algn: 32         used: 1
@7      integer_cst      type: @5         low : 55
@8      identifier_node  strg: void       lngt: 4
@9      type_decl        name: @14        type: @5
@10     integer_cst      type: @15        low : 32
@11     integer_cst      type: @5         high: -1         low : -2147483648
@12     integer_cst      type: @5         low : 2147483647
@13     identifier_node  strg: a          lngt: 1
@14     identifier_node  strg: int        lngt: 3
@15     integer_type     name: @16        size: @17        algn: 64
                         prec: 64         sign: unsigned   min : @18
                         max : @19
@16     identifier_node  strg: bit_size_type               lngt: 13
@17     integer_cst      type: @15        low : 64
@18     integer_cst      type: @15        low : 0
@19     integer_cst      type: @15        low : -1
```

Examples: GIMPLE dumps

2. gcc -fdump-tree-gimple test.c

test.c

```
int main()
{
        int a[3], x;
        a[1] = a[2] = 10;
         x = a[1] + a[2];
         a[0] = a[1] + a[1]*x;
}
```

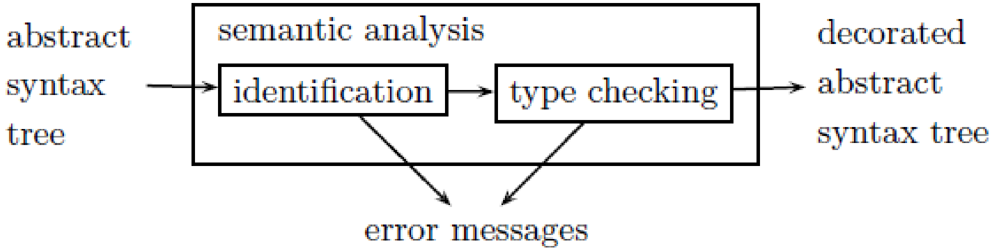
test.c.004t.gimple

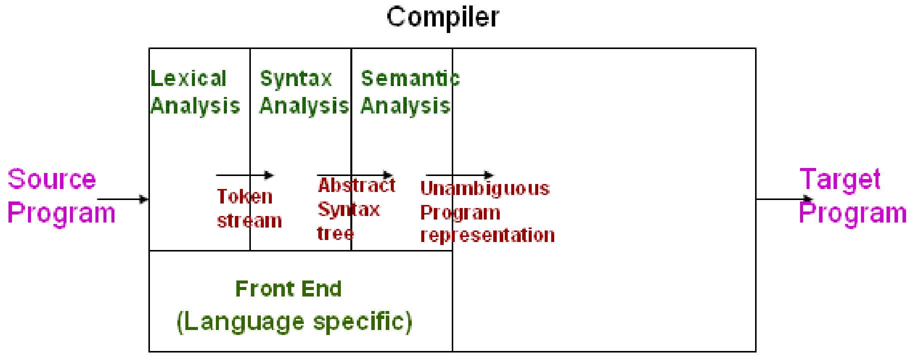
```
main ()
{
  int D.1589;
  int D.1590;
  int D.1591;
  int D.1592;
  int D.1593;
  int D.1594;
  int a[3];
  int x;

  a[2] = 10;
  D.1589 = a[2];
  a[1] = D.1589;
  D.1590 = a[1];
  D.1591 = a[2];
  x = D.1590 + D.1591;
  D.1592 = x + 1;
  D.1593 = a[1];
  D.1594 = D.1592 * D.1593;
  a[0] = D.1594;
}
```

Abstract Semantic Graph

- In computer science, an abstract semantic graph (ASG) or term graph is a form of abstract syntax in which an expression of a formal or programming language is represented by a graph whose vertices are the expression's sub-terms.
- An ASG is at a higher level of abstraction than an abstract syntax tree (or AST), which is used to express the syntactic structure of an expression or program.





Note: SSA, Decorated AST, ASG, Unambiguous Parse Tree are of similar meaning.

SSA

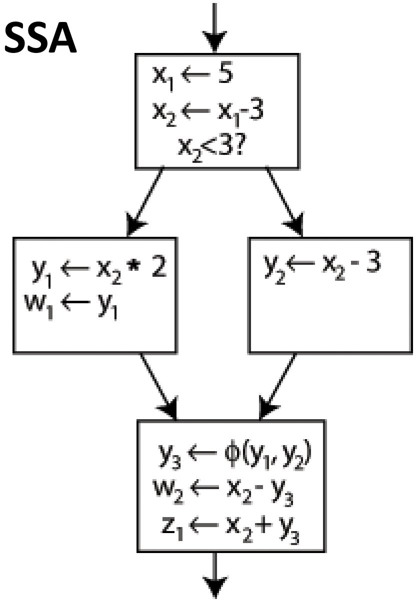
GIMPLE

```
a = 3;
b = 9;
c = a + b;
a = b + 1;
d = a + c;
return d;
```

SSA

```
a_1 = 3;
b_2 = 9;
c_3 = a_1 + b_2;
a_4 = b_2 + 1;
d_5 = a_4 + c_3;
_6 = d_5;
return _6;
```

SSA



Static Single Assignment

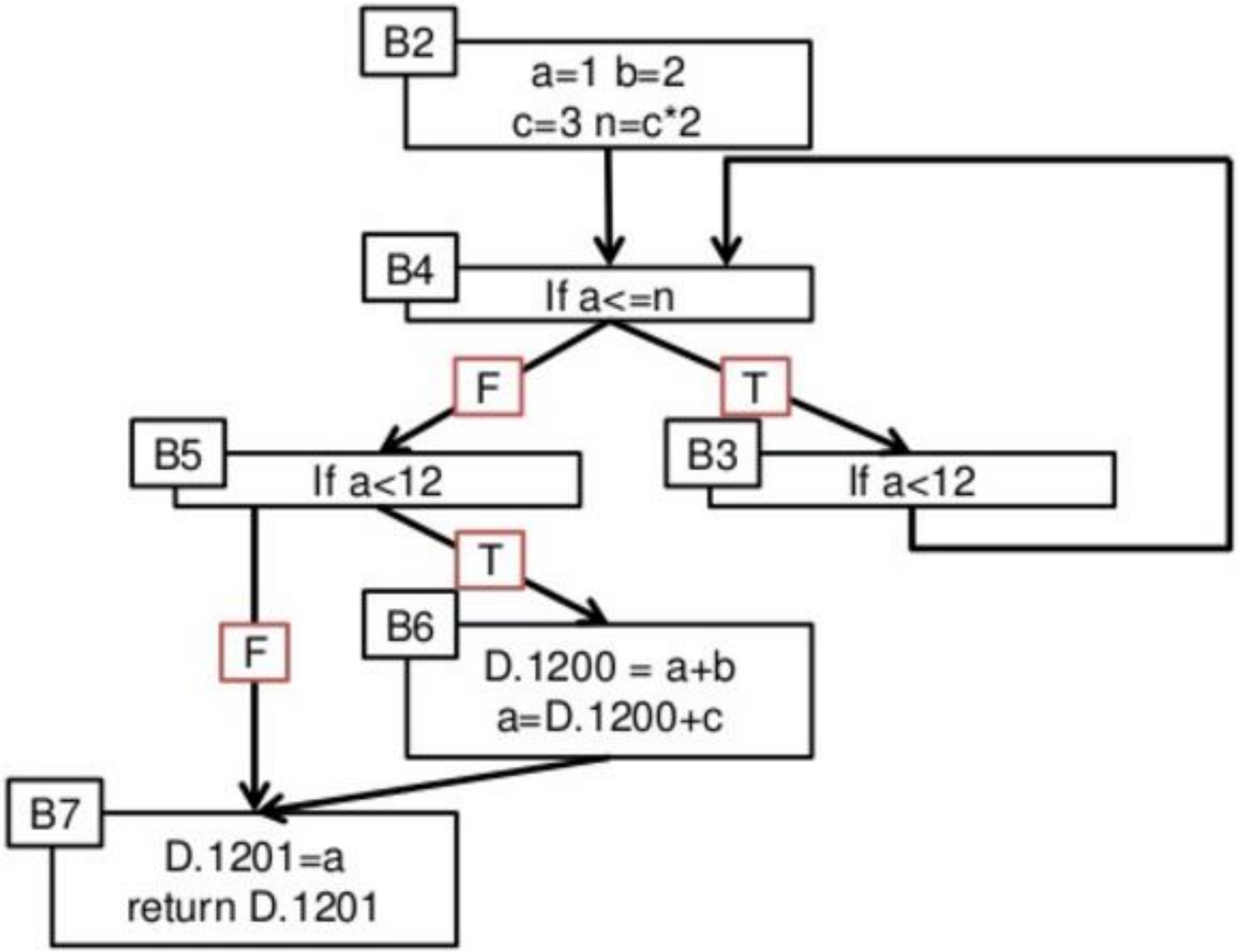
- Every variable is assigned only once
- Can be used as a read-only value multiple times
- In `if` statemens merging takes place
  - *PHI* function
- GCC performs over 20 optimizations on SSA tree

GCC Middle End

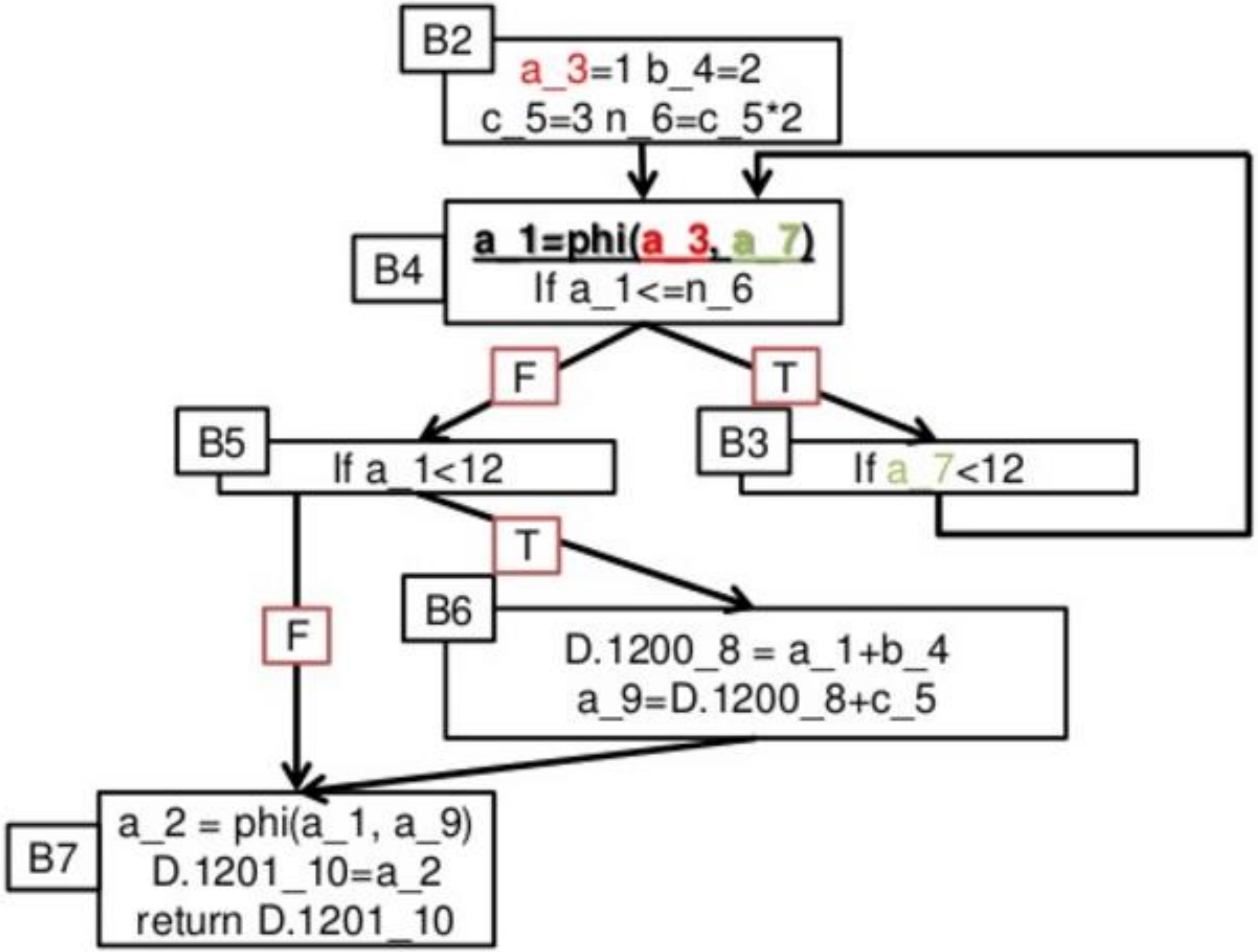
→ GIMPLE → SSA → Optimize → RTL →

- Generic → GIMPLE
- SSA transformation
- Optimization passes
- Un-SSA transformation
- RTL, suitable for back-end

cfg (Control Flow Graph)



cfg -> ssa (017t.ssa)



Examples: CFG dumps

3. gcc -fdump-tree-cfg test.c

test.c (part)

```
If (a<=12)
        a = a+b+c;
```

test.c.004t.gimple (part)

```
if (a<=12) goto
<D.1200>
else goto <D.1201>
<D.1200>:
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:
```

Examples: RTL dumps

4. gcc -fdump-rtl-expand test.c

test.c

```
int a;
main()
{
    a = a+1;
}
```

test.c.144r.expand (part)

```
(insn 5 4 6 3 (set (reg:SI 59 [ a.0 ])
        (mem/c/i:SI (symbol_ref:DI ("a")  <var_decl
0x7f13bdac3000 a>) [0 a+0 S4
 A32])) test.c:4 -1
     (nil))

(insn 6 5 7 3 (parallel [
            (set (reg:SI 60 [ a.1 ])
                (plus:SI (reg:SI 59 [ a.0 ])
                    (const_int 1 [0x1])))
            (clobber (reg:CC 17 flags))
        ]) test.c:4 -1
     (nil))

(insn 7 6 13 3 (set (mem/c/i:SI (symbol_ref:DI ("a")
<var_decl 0x7f13bdac3000 a>) [0 a+0 S4 A32])
        (reg:SI 60 [ a.1 ])) test.c:4 -1
     (nil))
```

Examples: Assembly dumps

5. gcc -S test.c || objdump -d a.out

test.c

```
int main()
{
int a;
a=1;
}
```

test.c.144r.expand (part)

```
(insn 5 4 11 3 (set (mem/c/i:SI (plus:DI (reg/f:DI
54 virtual-stack-vars)
                (const_int -4 [0xfffffffffffffffc])) [0 a+0
S4 A32])
        (const_int 1 [0x1])) test.c:4 -1
     (nil))
```

test.s (part)

```
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $1, -4(%rbp)
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

Role of Semantic Analysis

SECTION 3

Role of Semantic Analysis

- Following parsing, the next two phases of the "typical" compiler are
  - Semantic Analysis
  - (Intermediate) Code Generation
- The principal job of the semantic analyzer is to enforce static semantic rules
  - Constructs a Syntax Tree (usually first) GIMPLE/CFG/SSA
  - information gathered is needed by the code generator (SSA/IR+)

Role of Semantic Analysis

- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved
- A common approach interleaves construction of a syntax tree with parsing
  (no explicit parse tree), and then follows with separate, sequential phases for semantic analysis and code generation

Role of Semantic Analysis

- Semantic rules are further divided into static and dynamic semantics, though again the line between the two is somewhat fuzzy.
- The compiler enforces static semantic rules at compile time. It generates code to enforce dynamic semantic rules at runtime.
- Type checking and report errors at semantic analysis stage.
- Both semantic analysis and intermediate code generation can be described in terms of annotation, or decoration of parse tree. (**Decorated AST**).
- **Decoration** – Attributes.
- Most of this chapter is devoted to **Attribute Grammar**.

Attribute Grammar

Attribute grammars provide a formal framework for the decoration of a tree.

Topics in this chapter:

- **Attribute Grammar** framework is a useful conceptual tool even in compliers that do not build a parse tree or syntax tree as an explicit data structure.
- **Attribute Flow** constrains the order(s) in which nodes of a tree can be decorated. Most compilers required decoration of the parse tree to occur in the process of an LL or LR parse.
- **Action Routines** as an ad hoc mechanism for such “on-the-fly” evaluation.
- Management of the space for a parse tree.
- **Tree Grammar** and decoration of syntax tree.

Static versus Dynamic Semantics

- Static semantics
  - attribute grammars
- Dynamic semantics
  - operational semantics
  - axiomatic semantics
  - denotational semantics

Static vs. Dynamic

- We use the term *static* to describe properties that the compiler can determine without considering any particular execution.
  - E.g., in

    ```
    def f(x) : x + 1
    ```

    Both uses of x refer to same variable
- Dynamic properties are those that depend on particular executions in general.
  - E.g., will `x = x/y` cause an arithmetic exception?
- Actually, distinction is not that simple. E.g., after

  ```
  x = 3
  y = x + 2
  ```

  compiler *could* deduce that x and y are integers.
- But languages often designed to require that we treat variables only according to explicitly declared types, because deductions are difficult or impossible in general.

Dynamic Checks

- Many compilers that generate code for dynamic checks provide the option of disabling them if desired. [core dumps]
- It is customary in some organizations to enable dynamic checks during program development and test, and then disable them for production use, to increase execution speed.
- Errors may be less likely in production use than they are in testing, but the consequences of an undetected error are significantly worse.

Dynamic Checks

- On modern processors, it is often possible for dynamic checks to execute in pipeline slots that would otherwise go unused, making them virtually free.
- On the other hand, some dynamic checks are sufficiently expensive that they are rarely implemented.

Assertions in Java

- An assertions is a statement that a specified condition is expected to be true when execution reaches a certain point in the code.
- Assertions (by way of the assert keyword) were added in Java 1.4. They are used to verify the correctness of an invariant in the code. They should never be triggered in production code, and are indicative of a bug or misuse of a code path. They can be activated at run-time by way of the **-ea** option on the java command, but are not turned on by default.
- Replaced by JUnit Testing or Exception Handling

Assertions in Java

```java
public Foo acquireFoo(int id) {
  Foo result = null;
  if (id > 50) {
    result = fooService.read(id);
  } else {
    result = new Foo(id);
  }
  assert result != null;

  return result;
}
```

- An AssertionError exception will be thrown if the semantic check fails at run-time.

Assertion in C++

- Many languages support assertions via standard library routines or macros. In example, one can write:

```
assert(denominator != 0);
```

- If the assertion fails, the program will terminate abruptly with the message

```
myprog.c:42: failed assertion 'denominator != 0'
```

- The C manual requires assert to be implemented as a macro (or built into the compiler) so that it has access to the textual representation of its argument, and to the filename and line number on which the call appears.

Assertion in C++

- Assertions, of course, could be used to cover the other three sorts of checks, but not as clearly or succinctly. Invariants, preconditions, and post-conditions are a prominent part of the header of the code to which they apply, and can cover a potentially large number of places where an assertion would otherwise be required.
- Euclid and Eiffel implementations allow the programmer to disable assertions and related constructs when desired, to eliminate their run-time cost.

Static Analysis

In general, compile-time algorithms that predict run-time behavior are known as **static analysis**.

- **Type Checking:** mostly static but dynamically loaded classes and type casts may require run-time checks. Array subscripts, variant record tags, or dangling pointers.
- **Alias Analysis:** determines when values can be safely cached. [Cache Coherence]
- **Escape Analysis:** determines when all references to a value will be confined to a given context.

Static Analysis

- **Subtype Analysis:** determine a variable in OOP to have certain sub-type.
- **Optimization:**
  - Unsafe: lead to incorrect code
  - Speculative: improve performance but degrade sometimes.
  - Conservative: guarantee that it is safe and effective
  - Optimistic: make liberal use of speculative optimization.

To eliminate dynamic checking, language designer try to tighten the semantic rules.