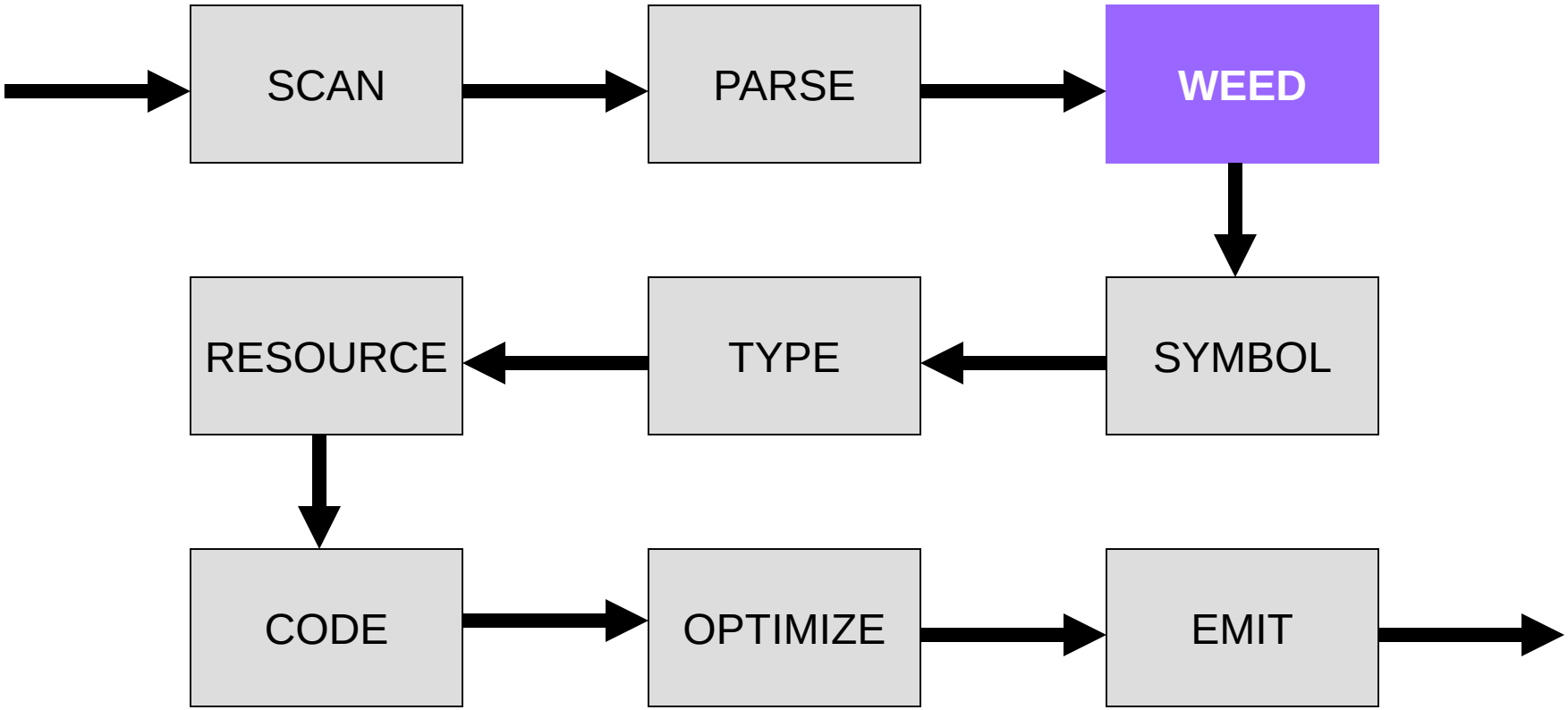
Compiler

The Abstract Syntax Tree (AST)

Abstract Syntax Trees



Phases and Passes

- A compiler *phase* is a cohesive functional unit that processes a representation of the source program
- A compiler *pass* is a traversal of the source program representation
- Multiple phases can be integrated into a single pass
  - e.g., parser-driven scanning

Single Pass Compilation

- All processing must happen in a pipelined fashion
  - Restricts languages (forward declarations)
  - Limits optimization
- Used to be popular:
  - fast (if your machine is slow); and
  - space efficient (if you only have 4K RAM)

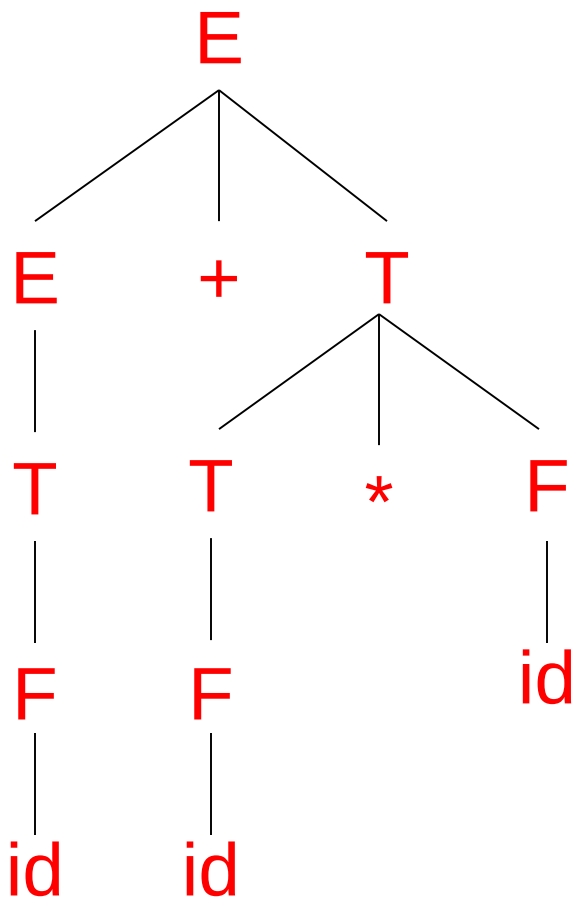
Multiple Pass Compilation

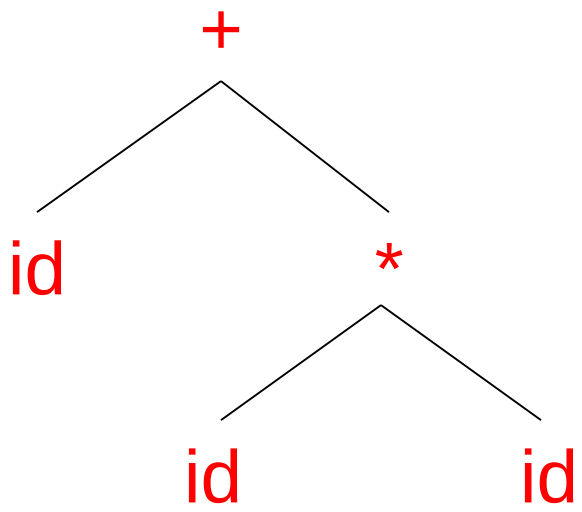
- *Speed/memory* is not a tight resource today
- A modern compiler uses 5-15 passes
- Advantage: Separation of concerns

Abstract Syntax Trees

- Common representation for programs
  - Represent all of the semantic entities in the program
  - Eliminates irrelevant syntactic details
- Can be thought of as a compressed parse tree
  - Need extra structure in the grammar during parsing for precedence, etc.
  - Don’t need all of that structure subsequently

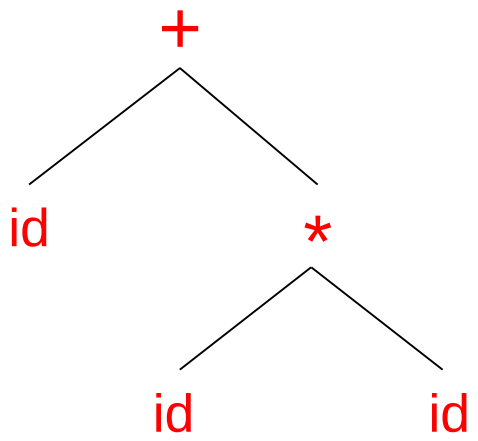
Parse Tree vs. AST





Intermediate Languages

- ASTs can be thought of as the input or output language of a phase
  - e.g., parser outputs ASTs, code generator takes AST as input
- *Linear forms* for ASTs allow phases to be written as separate programs
  - With their own parsers, etc.



+(id,*(id,id))

Enriching ASTs

- ASTs can carry lots of information about a program and its sub-parts

  Phases/passes may add their own info
  - Scanner ⟶ line numbers
    - Useful for error messages
  - Symbol processing ⟶ identifier meaning
  - Type checker ⟶ expression types
  - Code Generation ⟶ assembler code

AST Data Structure

- ANTLR provides a default AST structure
  - Allows arbitrary children using left-most child/right-sibling organization
  - Enabled by parser option **`output=AST`**
  - Will construct something like the parse tree by default
- Needs additional information
  - which tokens to be ignored
  - which tokens to be used as parent nodes
  - auxiliary tokens can be declared

ANTLR Rule Directives

- The ^ in a rule indicates the token used to determine the AST node that is built
- The ! in a rule indicates that the token should be ignored for the purpose of ASTs

Parse Actions

```
expr:   mexpr ((PLUS^|MINUS^) mexpr)*
    ;

mexpr
    :   atom (TIMES^ atom)*
    ;

atom:   INT^
    |   LPAREN! expr RPAREN!
    ;
```

ANTLR Rule Rewriting

- More powerfull and flexible than directives
  - Supports new node types (imaginary tokens)

```
 compilationUnit :
  packageDefinition? importDefinition* typeDefinition*
  -> ^(UNIT packageDefinition? importDefinition* typeDefinition*)
;
```

Imaginary Token Declaration

```
tokens
{
  UNIT; EXPR;
}
```

ANTLR Rule Rewriting

- More powerfull and flexible than directives
  - Supports new node types (imaginary tokens)
  - Allows decent restructuring

```
'if' '(' equalityExpression ')' s1=statement (
   'else' s2=statement )?
   -> ^('if' ^(EXPR equalityExpression) $s1 $s2)

 | -> ^('if' ^(EXPR equalityExpression) $s1)
);
```

- Much more details:

  http://www.antlr.org/wiki/display/ANTLR3/Tree+construction