

CS 321

## IV. Overview of Compilation

# Overview of Compilation

---

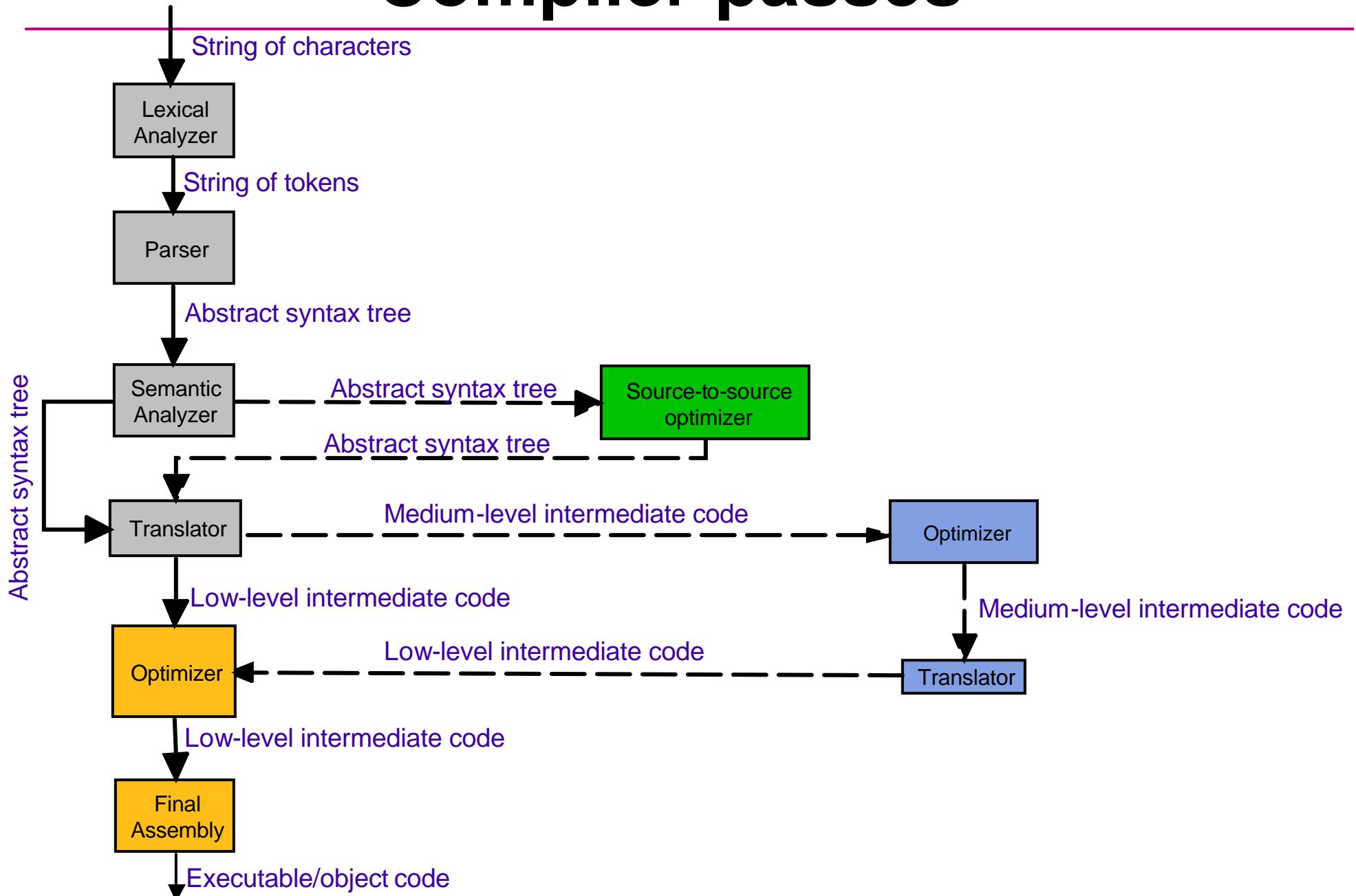
- Translating from high-level language to machine code is organized into several phases or passes.
- In the early days passes communicated through files, but this is no longer necessary.

# Language Specification

---

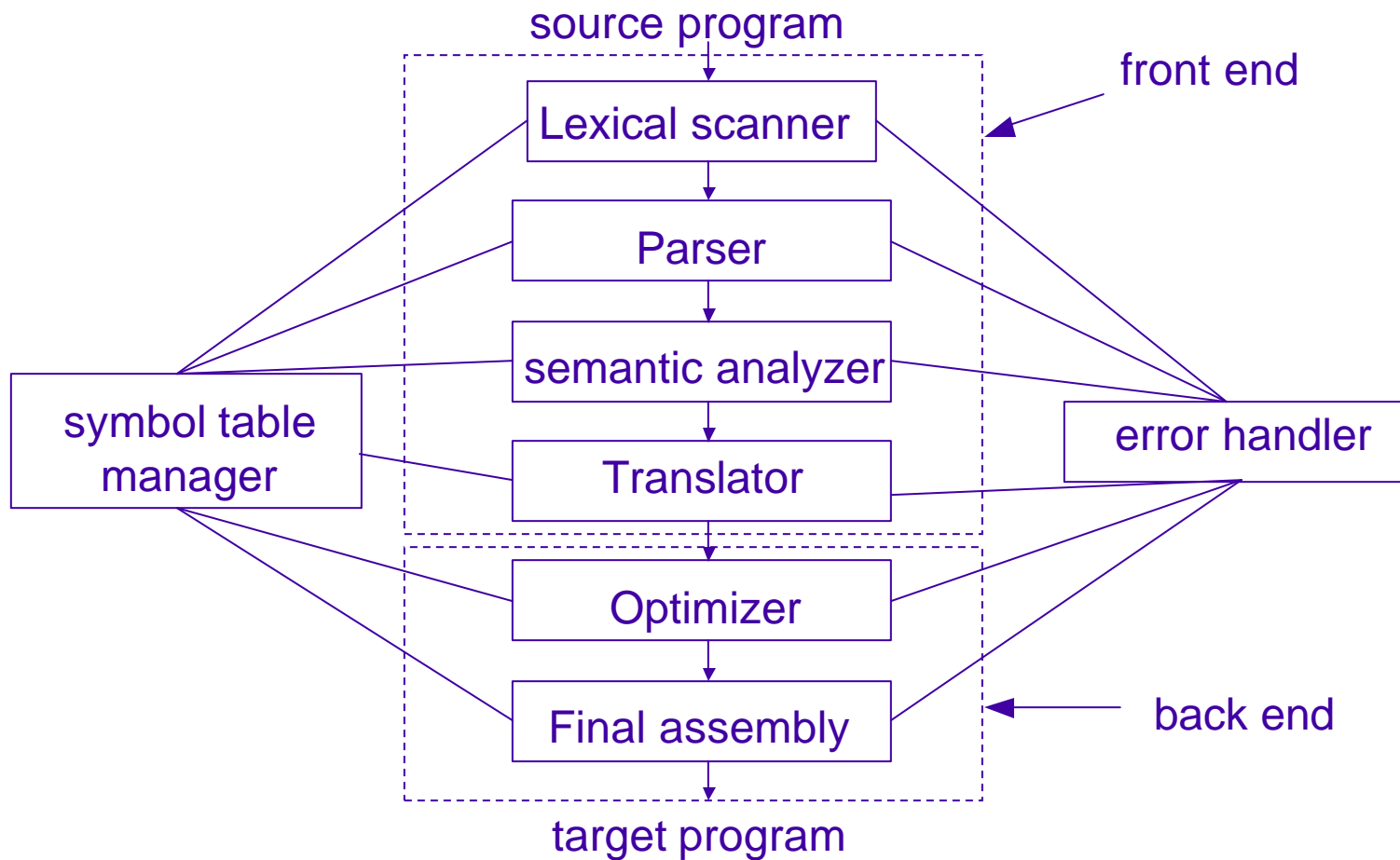
- We must first describe the language in question by giving its *specification*.
  - Syntax:
    - admissible symbols (vocabulary)
    - admissible programs (sentences)
  - Semantics:
    - give *meaning* to sentences.
- The formal specifications are often the input to tools that build translators automatically.

# Compiler passes



# Compiler passes

---



# Lexical analyzer

---

- A.k.a scanner or tokenizer.
- Converts stream of characters into a stream of tokens.
- Tokens are:
  - Keywords such as `for`, `while`, and `class`.
  - Special characters such as `+`, `-`, `(`, and `<`
  - Variable name occurrences
  - Constant occurrences such as `1.5e-10`, `true`.

# Lexical analyzer (Cont)

---

- The lexical analyzer is usually a subroutine of the parser.
- Each token is a single entity. A numerical code is usually assigned to each type of token.

- Lexical analyzers perform:
  - Line reconstruction
    - delete comments
    - delete extraneous blanks
    - handle margin checks
    - delete character/backspace
    - perform text substitution
  - Lexical translation: translation of *lexemes* -> *tokens*
    - + -> Taddop
    - >= -> Trelop
    - Joe -> Tident
    - Often additional information is affiliated with a token.

- Example. The program fragment

if num = 0 then avg := 0

else avg := sum \* num;

when “lexed” becomes

Tif Tid Trelop Tint Tthen Tid Tasgn Tint

Telse Tid Tasgn Tid Tmulop Tid Tsemi

- Keywords are easy to translate (finite functions).
- Relational Operators (and other “classes” of operators) are also easy to translate. Affiliated information identifies the specific operator.
- Identifiers, numbers, strings and the like are harder. There are an infinite number of them.
  - Finite State Machines model such tokens



- Grammars also model them

$$\text{id} \rightarrow aX \mid bX \mid \dots \mid zX$$

$$X \rightarrow aX \mid bX \mid \dots \mid zX \mid 0X \mid \dots \mid 9X \mid \varepsilon$$

# Parser

---

- Performs syntax analysis.
- Imposes syntactic structure on a sentence.
- Parse trees/derivation trees are used to expose the structure.
  - These trees are often not explicitly built.
  - Simpler representations of them are often used (e.g. syntax trees, tuples).
- Parser, accepts a string of tokens and builds a parse tree representing the program

# Parser (Cont.)

---

- The collection of all the programs in a given language is usually specified using a list of rules known as a context free grammar.
- In the next page we present a very simple grammar.

# Parser (Cont.)

---

program → **module identifier** { statement-list }

statement-list → statement ; statement-list

| statement

statement → **if** ( expression ) **then** { statement-list } **else** { statement-list }

| **identifier** = expression

| **while** ( expression < expression ) { statement-list }

| **input identifier**

| **output identifier**

expression → expression + term

| expression - term

| term

term → term \* factor

| term / factor

| factor

factor → ( expression )

| **identifier**

| **number**

# Parser (Cont.)

---

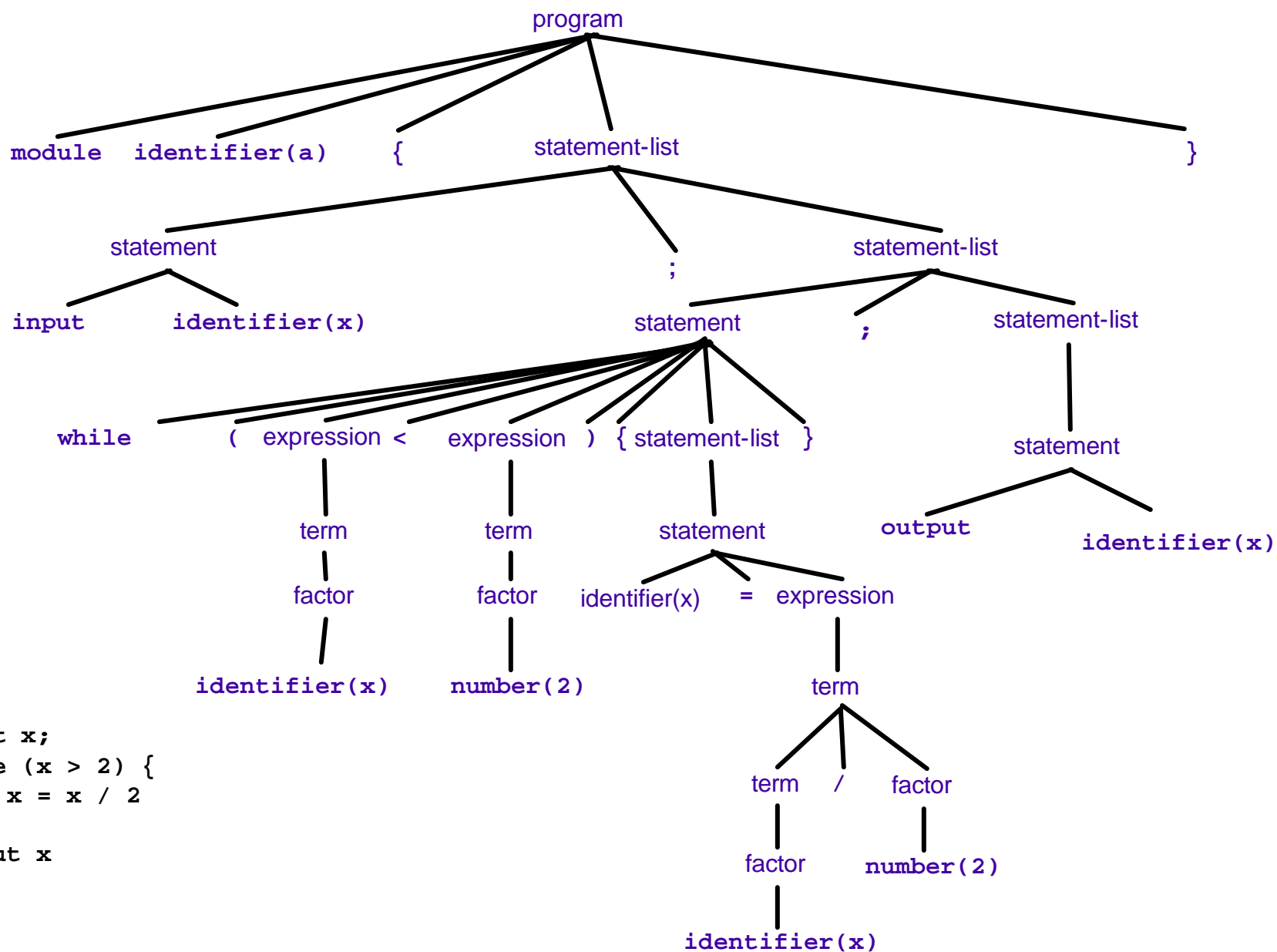
- A context free grammar like the one above has four components
  - A set of *tokens* known as *terminal* symbols
  - A set of *variables* or *nonterminals*
  - A set of *productions* where each production consists of a nonterminal, called the left side of the production, an arrow, and a sequence of tokens and/or nonterminals, called the right side of the production.
  - A designation of one of the nonterminals as the *start* symbol.

## Parser (Cont.)

---

- The terminals are represented in courier font in the previous example.
- The start symbol in the previous grammar is 'program'.
- A parse tree of a simple program using the grammar above is shown in the next page

# Parser (Cont.)



```

module a {
  input x;
  while (x > 2) {
    x = x / 2
  };
  output x
}
  
```

# Symbol Table Management

---

- The symbol table is a data structure used by all phases of the compiler to keep track of user defined symbols (and sometimes keywords as well).
- During early phases (lexical and syntax analysis) symbols are discovered and put into the symbol table
- During later phases symbols are looked up to validate their usage.

# Symbol Table Management (Cont.)

---

- Typical symbol table activities:
  - add a new name
  - add information for a name
  - access information for a name
  - determine if a name is present in the table
  - remove a name
  - revert to a previous usage for a name (close a scope).

# Symbol Table Management (Cont.)

---

- Many possible Implementations:
  - linear list
  - sorted list
  - hash table
  - tree structure

# Symbol Table Management (Cont.)

---

- Typical information fields:
  - print value
  - kind (e.g. reserved, typeid, varid, funcid, etc.)
  - block number/level number (in statically scoped languages)
  - type
  - initial value
  - base address
  - etc.

# Abstract Syntax Tree

---

- The parse tree is used to recognize the components of the program and to check that the syntax is correct.
- However, the parse tree is rarely built.
- As the parser applies productions, it usually generates the component of a simpler tree (known as Abstract Syntax Tree) that does not contain information that would be superfluous for the internal working of the compiler once a tree representing the program is built.
- For example, there is not need for the semi colon once each statement is organized as a subtree.

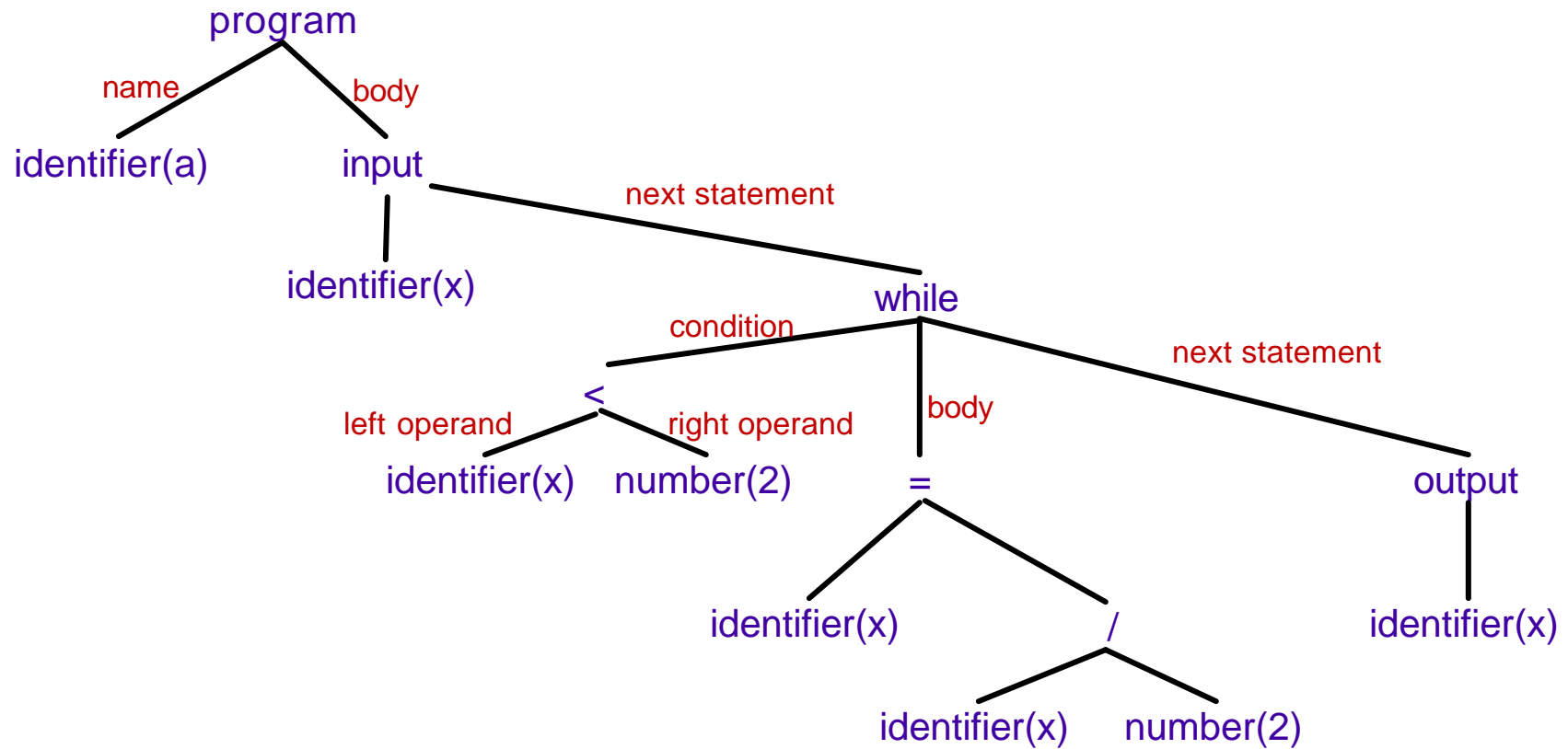
# Abstract Syntax Tree (Cont.)

---

- A typical Abstract Syntax Tree has labeled edges and follows the structure of the parse tree.
- The AST corresponding to the parse tree above is shown next.

# Abstract Syntax Tree (Cont.)

---



# Semantic Analyzer

---

- The semantic analyzer completes the symbol table with information on the characteristics of each identifier.
- The symbol table is usually initialized during parsing. One entry is created for each identifier and constant.
  - Scope is taken into account. Two different variables with the same name will have different entries in the symbol table.
- Nodes in the AST containing identifiers or constants really contain pointers into the symbol table.
- The semantic analyzer completes the table using information from declarations and perhaps the context where the identifiers occur.

# Semantic Analyzer (Cont.)

---

- The semantic analyzer does
  - Type checking
  - Flow of control checks
  - Uniqueness checks (identifiers, case labels, etc.)
- One objective is to identify semantic errors statically. For example:
  - Undeclared identifiers
  - Unreachable statements
  - Identifiers used in the wrong context.
  - Methods called with the wrong number of parameters or with parameters of the wrong type.
  - ...

# Semantic Analyzer (Cont.)

---

- Some semantic errors have to be detected at run time to make sure they are always caught. The reason is that the information may not be available at compile time.
  - Array subscript is out of bounds.
  - Variables are not initialized.
  - Divide by zero.
- Notice, however, that in some cases (but not always) these semantic errors can be detected statically.

# Semantic Analyzer (Cont.)

---

For example, it is possible to detect statically (at compile-time) that **a** is never out of bounds in the loop

```
float a[ ]=new float[100]
for(i=1;i<20;i++){ a[i]=0 }
```

However, static analysis would not work if the loop has the following form

```
float a[ ]=new float[100]
n=read ();
for(i=1;i<n;i++){ a[i]=0 }
```

# Error Management

---

- Errors can occur at all phases in the compiler
- Invalid input characters, syntax errors, semantic errors, etc.
- Good compilers will attempt to recover from errors and continue.

# Source-to-source optimizers

---

- It is possible to do code improvement at the high-level ( or source-to-source transformations).
- An example of such transformation is the locality enhancement transformation presented earlier.

# Translator

---

- The lexical scanner, parser, semantic analyzer, and source-to-source optimizer are collectively known as the *front end* of the compiler.
- The second part, or *back end* starts by generating low level code from the (possibly optimized) AST.

# Translator

---

- Rather than generate code for a specific architecture, most compilers generate intermediate language
- Three address code is popular.
  - Really a flattened tree representation.
  - Simple.
  - Flexible (captures the essence of many target architectures).
  - Can be interpreted.

# Translator

---

- One way of performing intermediate code generation:
  - Attach meaning to each node of the AST.
  - The meaning of the sentence = the “meaning” attached to the root of the tree.

# XIL

---

- An example of Medium level intermediate language is XIL. XIL is used by IBM to compile FORTRAN, C, C++, and Pascal for RS/6000.
- Compilers for Fortran 90 and C++ have been developed using XIL for other machines such as Intel 386, Sparc, and S/370.
- These compiler share a common back end, the Toronto Optimizing Back End with Yorktown, TOBEY.

# XIL (Cont.)

---

- XIL is free of source language dependences and thus forms a suitable target for the compilation of a broad range of programming languages.
- XIL presents a model of a machine with a Load/Store architecture and a number of distinct register sets.
- These register sets can each contain an unbounded number of symbolic registers.
- Instructions are similar to those of RISC machines, but more flexible:
  - displacement in addresses can be of any size
  - addresses can contain as many index registers as desired
  - call instructions can have a large number of parameter registers
  - instructions can have as many result registers as is convenient

# XIL (Cont.)

- Indexed load of  $a(i, j+2)$  in XIL

```
L   r.i=i(r200,64)
```

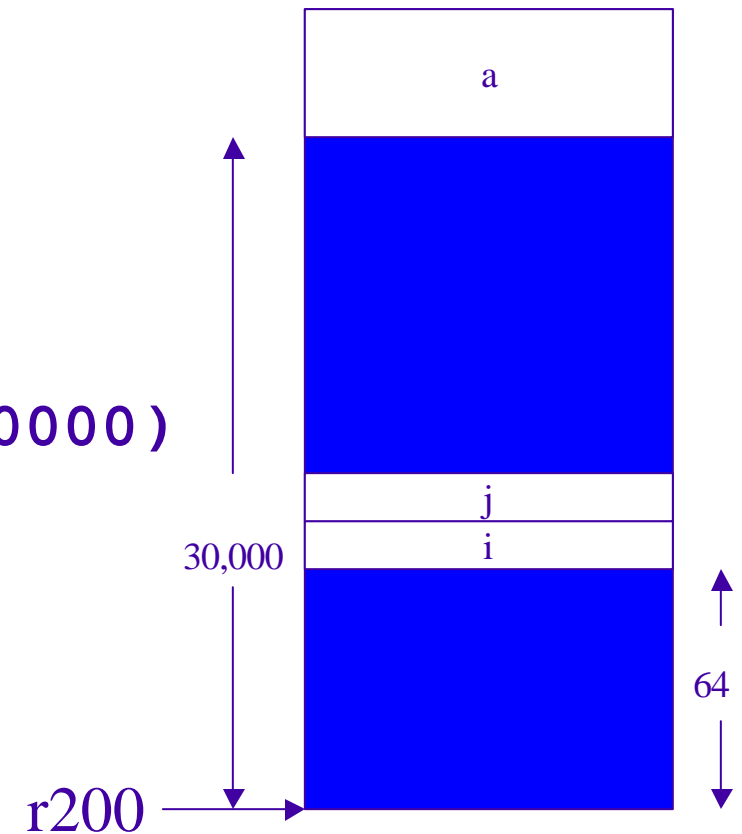
```
M   r300=r.i,8
```

```
L   r.j=j(r200,68)
```

```
A   r310=r.j,2
```

```
M   r320=r310,400
```

```
LFL fp330=a(r200,r300,r320,30000)
```



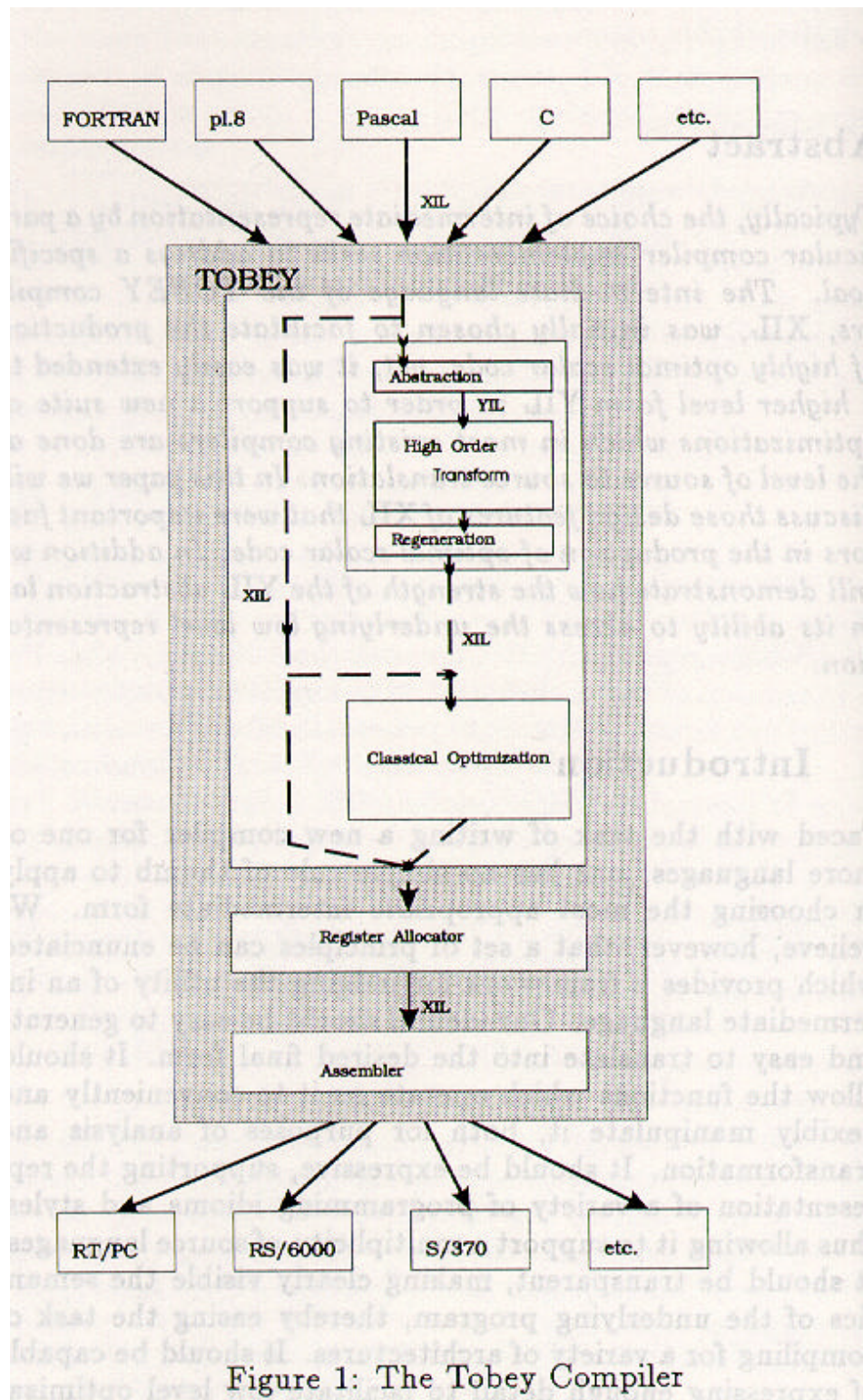


Figure 1: The Tobey Compiler

# Optimizers

---

- Improve the quality of the code generated.
- Optimizers manipulating Medium level intermediate language apply machine-independent transformations.
  - Loop invariant removal
  - Common subexpression elimination
- Optimizers manipulating low level language apply machine-dependent transformations.
  - Register allocation

# Optimizers (Cont.)

---

- Intermediate code is examined and improved.
- Can be simple:
  - changing “`a := a + 1`” to “increment `a`”
  - changing “`3 * 5`” to “`15`”
- Can be complicated:
  - reorganizing data and data accesses for cache efficiency
- Optimization can improve running time by orders of magnitude, often also decreasing program size.

# Code Generation

---

- Generation of “real executable code” for a particular target machine.
- It is completed by the Final Assembly phase, but other passes contribute to it
- Final output can either be
  - assembly language for the target machine (requiring execution of the assembler after the compiler completes)
  - object code ready for linking
- The “target machine” can be a virtual machine (such as the Java Virtual Machine, JVM), and the “real executable code” is “virtual code” (such as Java Bytecode).

# Tools

---

- Tools exist to help in the development of some stages of the compiler
- Lex (Flex) - lexical analyzer generator
- Yacc (Bison) - parser generator