LECTURE 3

Compiler Phases

COMPILER PHASES

Compilation of a program proceeds through a fixed series of phases.

- Each phase uses an (intermediate) form of the program produced by an earlier phase.
- Subsequent phases operate on lower-level code representations.

Each phase may consist of a number of passes over the program representation.

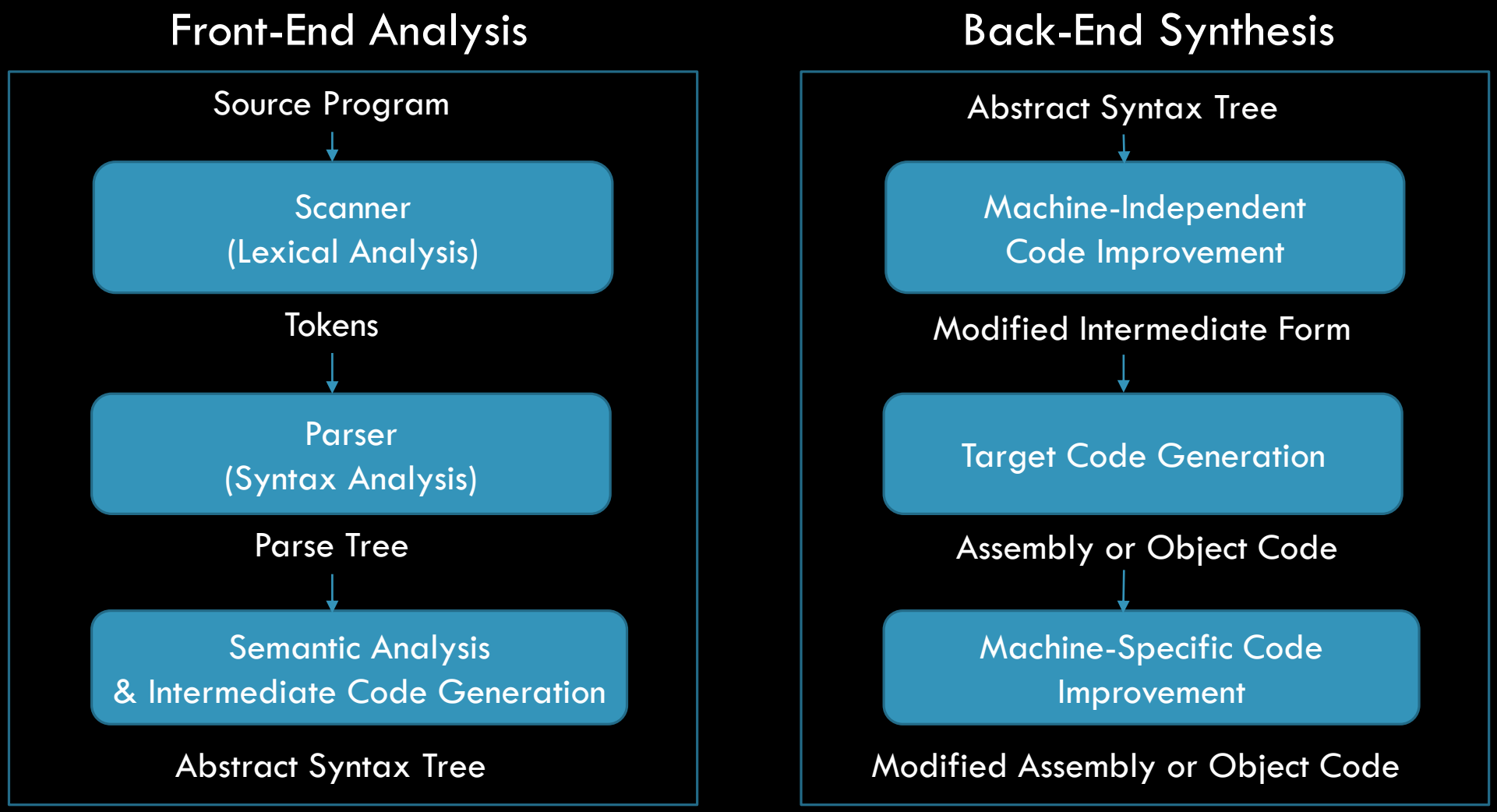
- Pascal, FORTRAN, and C languages designed for one-pass compilation, which explains the need for function prototypes.
- Single-pass compilers need less memory to operate.
- Java and Ada are multi-pass.

COMPILER PHASES

The compiling process phases are listed below.

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate (machine-independent) code generation
- Intermediate code optimization
- Target (machine-dependent) code generation
- Target code optimization

FRONT- AND BACK-END



LEXICAL ANALYSIS

Lexical analysis is the process of *tokenizing* characters that appear in a program.

A *scanner* (or *lexer*) groups characters together into meaningful tokens which are then sent to the *parser*.

What we write:

```
int main(){
    int i = getint(), j = getint();
    while(i!=j){
        if(i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```

What the scanner picks up:

‘i’, ‘n’, ‘t’, ‘ ’, ‘m’, ‘a’, ‘i’, ‘n’, ‘(’, ‘)’, ‘{’….

LEXICAL ANALYSIS

As the scanner reads in the characters, it produces meaningful tokens.

Tokens are typically defined using *regular expressions*, which are understood by a lexical analyzer generator such as lex.

What the scanner picks up:

‘i’, ‘n’, ‘t’, ‘ ’, ‘m’, ‘a’, ‘i’, ‘n’, ‘(’, ‘)’, ‘{’….

The resulting tokens:

int, main, (,), {, int, i, =, getint, (,), ….

LEXICAL ANALYSIS

What kinds of errors can be caught in the lexical analysis phase?

LEXICAL ANALYSIS

What kinds of errors can be caught in the lexical analysis phase?

- Invalid tokens.
- An example in C: `int i = @3;`

@ is an invalid token in C

SYNTAX ANALYSIS

Syntax analysis is performed by a *parser* which takes the tokens generated by the scanner and creates a *parse tree* which shows how tokens fit together within a valid program.

The structure of the parse tree is dictated by the grammar of the programming language.

- Typically, a *context-free grammar* which is a set of recursive rules for generating valid patterns of tokens.

CONTEXT-FREE GRAMMARS

Context-free grammars define the syntax of a programming language.

Generally speaking, context-free grammars contain recursive rewrite rules (*productions*) of the following form:

$$N \rightarrow w$$

where N is a *non-terminal* symbol and w is a string of *terminal* symbols and/or *non-terminal* symbols. Alternatively, w could also be ϵ, an empty string.

CONTEXT-FREE GRAMMARS

Here's a little example. The starting non-terminal is *program*.

program	→	*statement*
statement	→	if (*condition*) { *statement* }
statement	→	*assignment*
assignment	→	*exp* = *exp* ;
condition	→	*exp* > *exp*
condition	→	*exp* < *exp*
exp	→	id

CONTEXT-FREE GRAMMARS

What kind of code does this CFG admit?

program	→	*statement*
statement	→	if (*condition*) { *statement* }
statement	→	*assignment*
assignment	→	*exp* = *exp* ;
condition	→	*exp* > *exp*
condition	→	*exp* < *exp*
exp	→	id

```
a = b;
```

```
if(a > b){
    b = c;
}
```

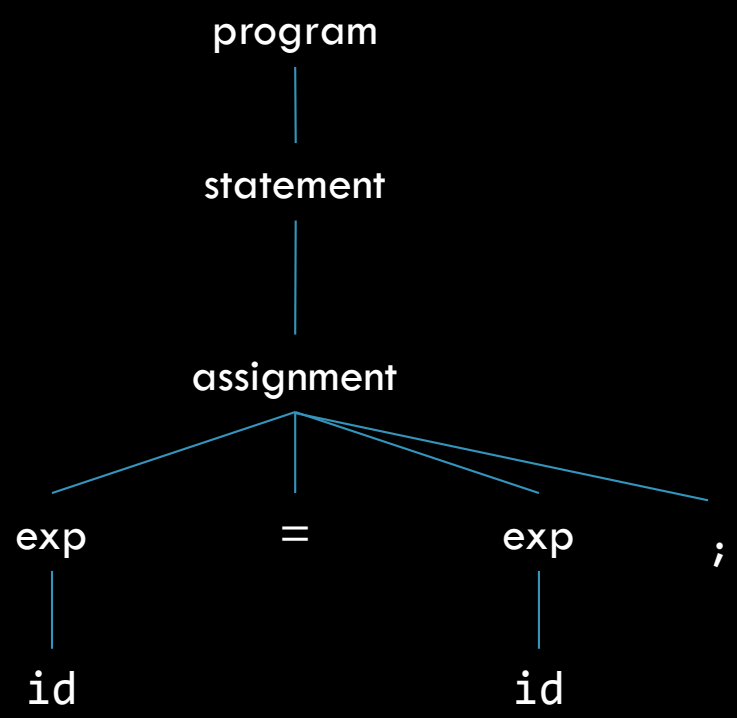
```
if(a > b){
    if(a > c){
        b = d;
    }
}
```

SYNTAX ANALYSIS

How can we use our CFG to create the parse tree?

program	→	*statement*
statement	→	if (*condition*) { *statement* }
statement	→	*assignment*
assignment	→	*exp* = *exp* ;
condition	→	*exp* > *exp*
condition	→	*exp* < *exp*
exp	→	id

```
a = b;
```

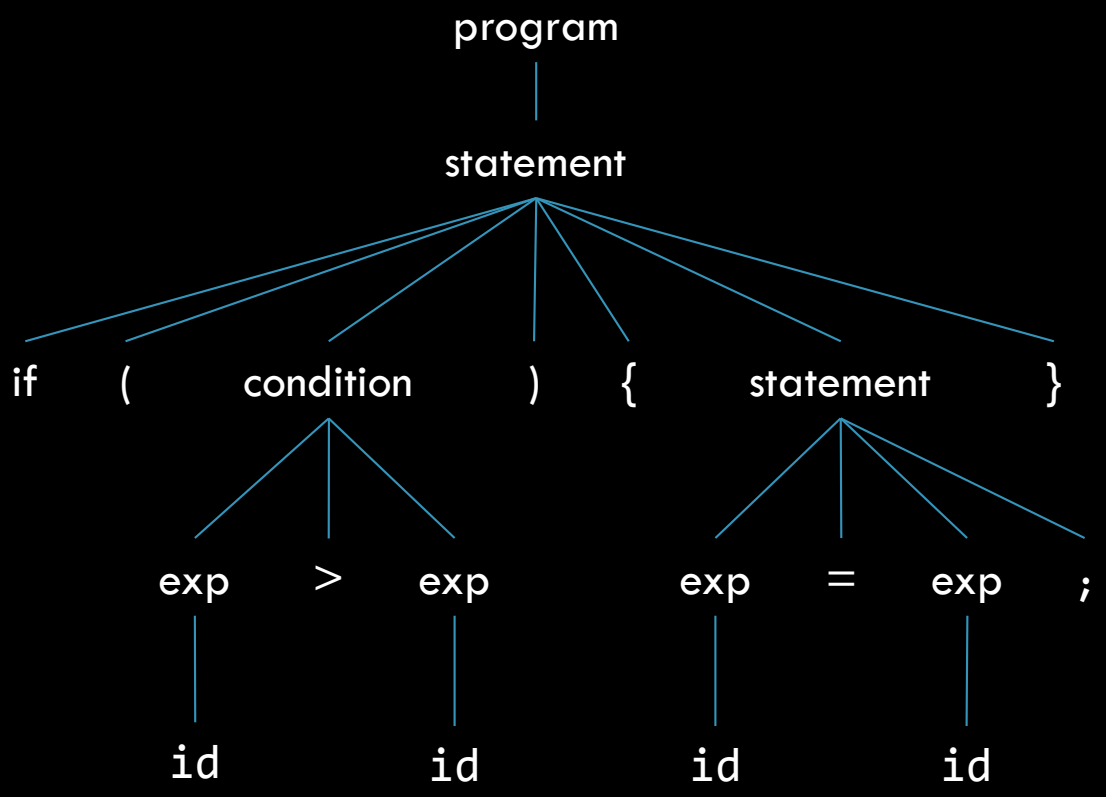


SYNTAX ANALYSIS

A more complicated example…

program	→	*statement*
statement	→	if (*condition*) { *statement* }
statement	→	*assignment*
assignment	→	*exp* = *exp* ;
condition	→	*exp* > *exp*
condition	→	*exp* < *exp*
exp	→	id

```
if(a > b){
    b = c;
}
```



SYNTAX ANALYSIS

What kinds of errors are caught by the parser?

SYNTAX ANALYSIS

What kinds of errors are caught by the parser?

- Syntax errors: invalid sequences of tokens.
- Example in C: `a * b = c;`

SEMANTIC ANALYSIS

Semantic analysis is the process of attempting to discover whether a valid pattern of tokens is actually meaningful.

Even if we know that the sequence of tokens is valid, it may still be an incorrect program.

For example: `a = b;`

What if `a` is an int and `b` is a character array?

To protect against these kinds of errors, the semantic analyzer will keep track of the types of identifiers and expressions in order to ensure they are used consistently.

SEMANTIC ANALYSIS

- Static Semantic Checks: semantic rules that can be checked at compile time.
  - Type checking.
  - Every variable is declared before being used.
  - Identifiers are used in appropriate contexts.
  - Checking function call arguments.
- Dynamic Semantic Checks: semantic rules that are checked at run time.
  - Array subscript values are within bounds.
  - Arithmetic errors, e.g. division by zero.
  - Pointers are not dereferenced unless pointing to valid object.
  - When a check fails at run time, an exception is raised.

SEMANTIC ANALYSIS

Semantic analysis is typically performed by creating a symbol table that stores necessary information for verifying consistency.

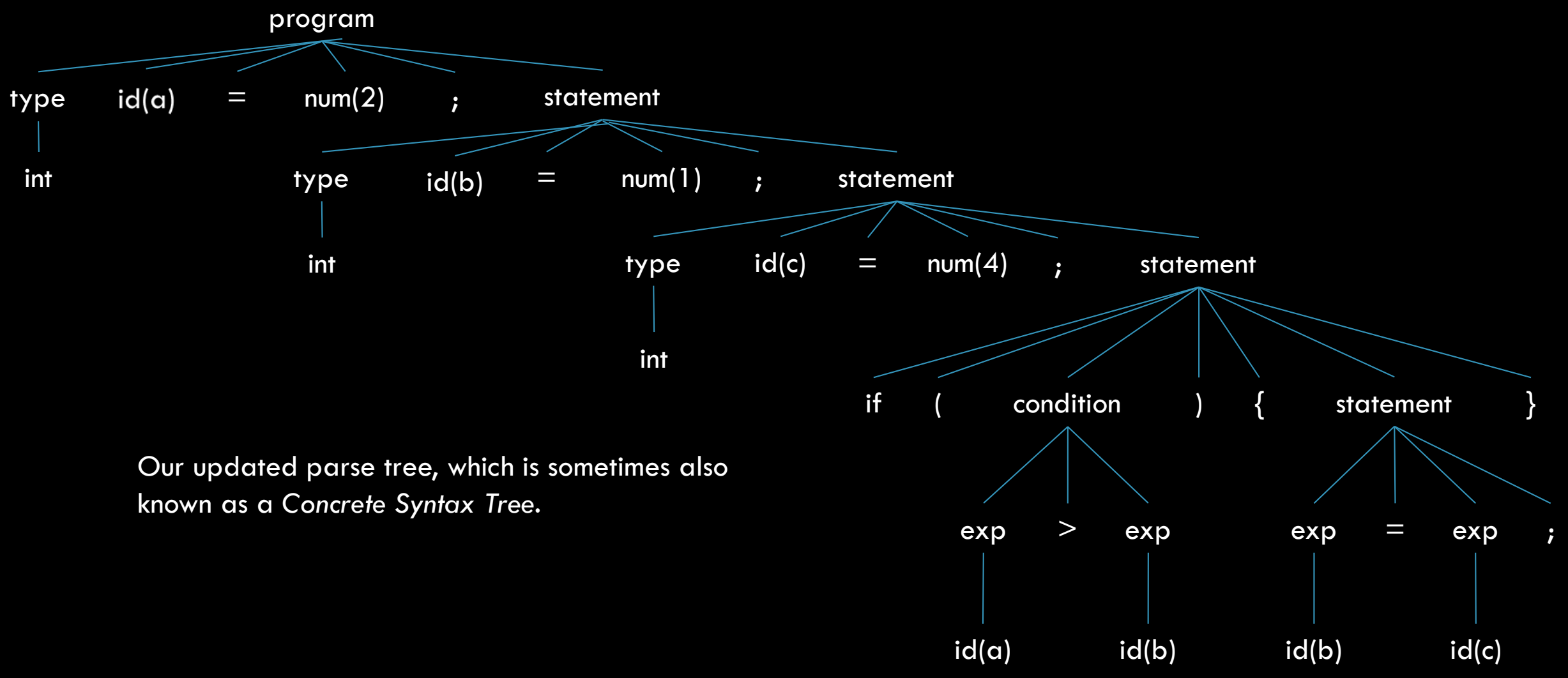
A typical intermediate form of code produced by the semantic analyzer is an Abstract Syntax Tree (AST), which is annotated with pointers to the symbol table.

Let's look at our previous example and add some more rules to make it viable.

```
int a = 2;
int b = 1;
int c = 4;
if(a > b){
    b = c;
}
```

program → *statement*
statement → if (*condition*) { *statement* }
statement → *assignment*
statement → *type exp* = num ; *statement*
assignment → *exp* = *exp* ;
condition → *exp* > *exp*
condition → *exp* < *exp*
exp → id
type → int

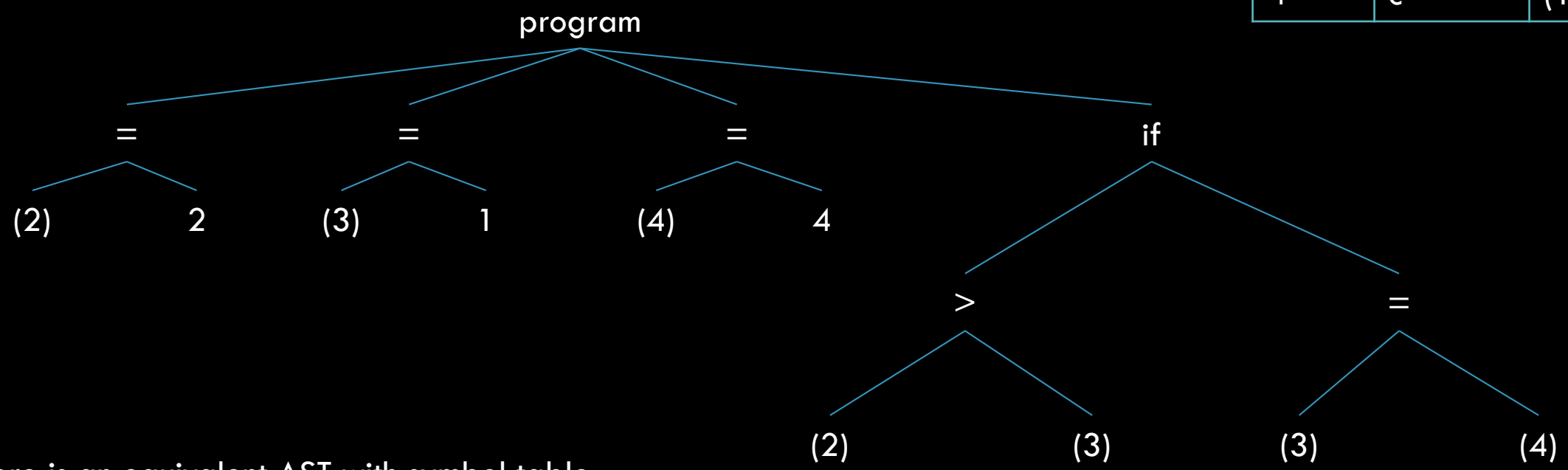
SEMANTIC ANALYSIS



Our updated parse tree, which is sometimes also known as a *Concrete Syntax Tree*.

SEMANTIC ANALYSIS

Index	Symbol	Type
1	int	type
2	a	(1)
3	b	(1)
4	c	(1)



Here is an equivalent AST with symbol table.
The goal of the AST is to produce an intermediate form of the code
that removes the “artificial” nodes and annotates the rest with useful information.
Some compilers use alternative intermediate forms, but the goal is the same.

SYNTAX AND SEMANTIC ANALYSIS

Assuming C++, what kinds of errors are these (lexical, syntax, semantic)?

- int = @3;
- int = ?3;
- int y = 3; x = y;
- “Hello, World!
- int x; double y = 2.5; x = y;
- void sum(int, int); sum(1,2,3);
- myint++
- z = y/x; // y is 1, x is 0

SYNTAX AND SEMANTIC ANALYSIS

Assuming C++, what kinds of errors are these (lexical, syntax, semantic)?

- int = @3; // Lexical
- int = ?3; // Syntax
- int y = 3; x = y; // Static Semantic
- “Hello, World! // Syntax
- int x; double y = 2.5; x = y; // Static Semantic
- void sum(int, int); sum(1,2,3); // Static Semantic
- myint++ // Syntax
- z = y/x; // y is 1, x is 0 // Dynamic Semantic

CODE OPTIMIZATION

Once the AST (or alternative intermediate form) has been generated, the compiler can perform machine-independent code optimization.

The goal is to modify the code so that it is quicker and uses resources more efficiently.

There is an additional optimization step performed after the creation of the object code.

TARGET CODE GENERATION

The goal of the Target Code Generation phase is to translate the intermediate form of the code (typically, the AST) into object code.

In the case of languages that translate into assembly language, the code generator will first pass through the symbol table, creating space for the variables.

Next, the code generator passes through the intermediate code form, generating the appropriate assembly code.

As stated before, the compiler makes one more pass through the object code to perform further optimization.

NEXT LECTURE

Regular Expressions and Context-Free Grammars