Question of the Day

Backpatching

`o.foo();`

In Java, the address of `foo()` is often not known until runtime (due to dynamic class loading), so the method call requires a table lookup.

After the first execution of this statement, backpatching replaces the table lookup with a direct call to the proper function.

Q: How could backpatching ever hurt?

A: The Pentium 4 has a trace cache, when any instruction is modified, the entire trace cache has to be flushed.

Undergraduate Compilers in a Day

Today

- Leftovers from last lecture
- Overall structure of a compiler
- Intermediate representations

- Focus on traditional imperative languages

Leftovers from Last Class

Phase Ordering Problem

In what order should optimizations be performed?

Simple dependences

- One optimization creates opportunity for another
  e.g., copy propagation and dead code elimination

Cyclic dependences

- *e.g.*, constant folding and constant propagation

Adverse interactions

- *e.g.*, common sub-expression elimination and register allocation
  e.g., register allocation and instruction scheduling

Engineering Issues

Building a compiler is an engineering activity

Balance multiple goals

- Benefit for *typical* programs
- Complexity of implementation
- Compilation speed

Overall Goal

- Identify a small set of general analyses and optimization
- Easier said than done: just one more...

Workload

The workload is heavy

– Facility with C++ is important

Academic Dishonesty

Consequence

- Cheating will lead to failure of the course

If you have any questions, ask

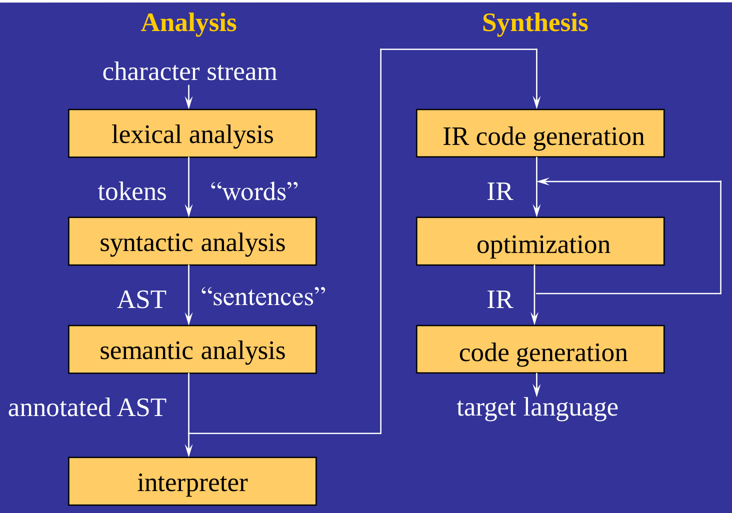
Undergraduate Compilers in a Day

Today

- Leftovers from last lecture
- Overall structure of a compiler
- Intermediate representations

- Focus on traditional imperative languages

Structure of a Typical ~~Interpreter~~ Compiler



Analysis

Synthesis

character stream

lexical analysis

tokens

“words”

syntactic analysis

AST

“sentences”

semantic analysis

annotated AST

interpreter

IR code generation

IR

optimization

IR

code generation

target language

Lexical Analysis (Scanning)

Break character stream into tokens (“words”)

- Tokens, lexemes, and patterns
- Lexical analyzers (*e.g.*, lex) are usually automatically generated from patterns (regular expressions)

token	**lexeme(s)**	**pattern**
const	`const`	`const`
if	`if`	`if`
relation	`<,<=,=,!=,...`	`< | <= | = | != | ...`
identifier	`foo,index`	`[a-zA-Z_]+[a-zA-Z0-9_]*`
number	`3.14159,570`	`[0-9]+ | [0-9]*.[0-9]+`
string	`“hi”, “mom”`	`“.*”`

Examples

`const pi := 3.14159` ⇒ *const*, *identifier*(`pi`), *assign*,*number*(**3.14159**)

Syntactic Analysis (Parsing)

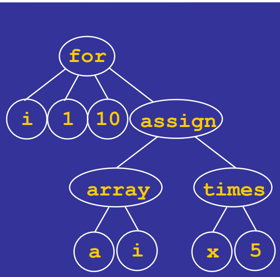
Impose structure on token stream

- Limited to syntactic structure (⇒ high-level)
- Structure usually represented with an *abstract syntax tree (AST)*
- Theory meets practice:
  - Regular expressions, formal languages, grammars, parsing…
- Parsers are usually automatically generated from grammars (*e.g.*, yacc, bison, cup, javacc)

Syntactic Analysis (Parsing)

Example

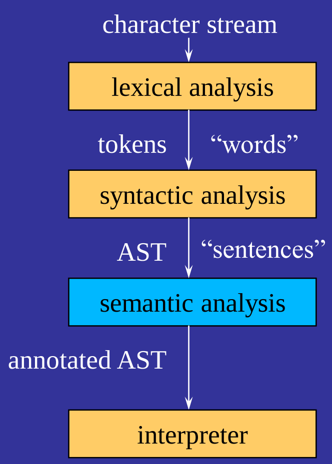
```
for i = 1 to 10 do
    a[i] = x * 5;
```



for id(i) *equal number*(1) *to number*(10) *do*
id(a) *lbracket id*(i) *rbracket equal id*(x) *times number*(5) *semi*

Structure of a Typical Interpreter

Analysis



Semantic Analysis

Determine whether source is meaningful

- Check for semantic errors
- Check for type errors
- Gather type information for subsequent stages
  - Relate variable uses to their declarations
- Some semantic analysis takes place during parsing

Example errors (from C)

```
function1 = 3.14159;
x = 570 + “hello, world!”
scalar[i];
```

Compiler Data Structures

Symbol Tables

- Compile-time data structures
- Hold names, type information, and *scope* information for variables

Scopes

- A name space
  e.g., In Pascal, each procedure creates a new scope
  e.g., In C, each set of curly braces defines a new scope
- Can create a separate symbol table for each scope

Compiler Data Structures (cont)

Using Symbol Tables

- For each variable declaration:
  - Check for symbol table entry
  - Add new entry (parsing); add type info (semantic analysis)

- For each variable use:
  - Check symbol table entry (semantic analysis)

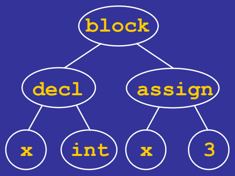
Exercise: Symbol Table Alternative

Idea

- Dispense with explicit symbol table structure
- Include declarations in AST

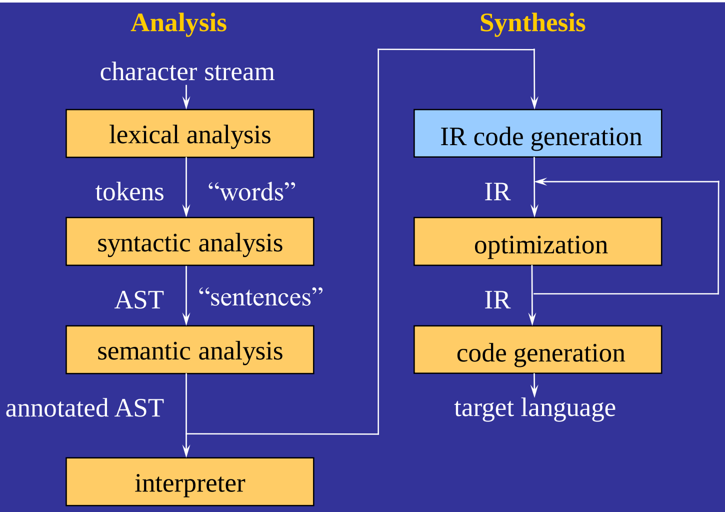
Example

```
{
  int x;
  x = 3;
}
```



Discuss the advantages and disadvantages of this idea

Structure of a Typical Compiler



Analysis

character stream

lexical analysis

tokens “words”

syntactic analysis

AST “sentences”

semantic analysis

annotated AST

interpreter

Synthesis

IR code generation

IR

optimization

IR

code generation

target language

IR Code Generation

Goal

- Transform AST into low-level *intermediate representation* (IR)

Simplifies the IR

- Removes high-level control structures:

  `for`, **`while`**, **`do`**, **`switch`**
- Removes high-level data structures:

  arrays, structs, unions, enums

IR Code Generation (cont)

One possible result is assembly-like code

- Semantic lowering
- Control-flow expressed in terms of “gotos”
- Each expression is very simple (three-address code)

  e.g., `x := a * b * c` ➡ `t := a * b`

  `x := t * c`

A Low-Level IR

Register Transfer Language (RTL)

- Linear representation
- Typically language-independent
- Nearly corresponds to machine instructions

Example operations

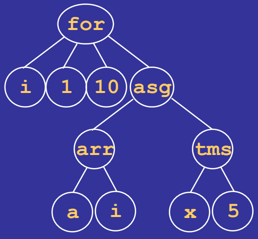
- Assignment `x := y`
- Unary op `x := op y`
- Binary op `x := y op z`
- Call `x := f()`
- Cbranch `if (x==3) goto L1`
- Address of `p := & y`
- Load `x := *(p+4)`
- Store `*(p+4) := y`

Example

Source code

```
for i = 1 to 10 do
   a[i] = x * 5;
```

High-level IR (AST)



Low-level IR (RTL)

```
         i  := 1
loop1:
        t1  := x * 5
        t2  := &a
        t3  := sizeof(int)
        t4  := t3 * i
        t5  := t2 + t4
       *t5  := t1
         i  := i + 1
        if i <= 10 goto loop1
```

Exercise

High-level IR vs. Low-Level IR?

Next Time

Lecture

- Control flow analysis

Assignment 0

- Due Wednesday

Assignment 1

- Familiarize yourself with the LLVM compiler
- Due next Monday (February 2)