CS302 Languages and Compiler Design II

Lecture 1

Languages and Compiler Design Part II

Topics

- Semantics
- Interpreters
- Runtime Organization
- Intermediate Code Generation
- Machine Code Generation
- Optimization

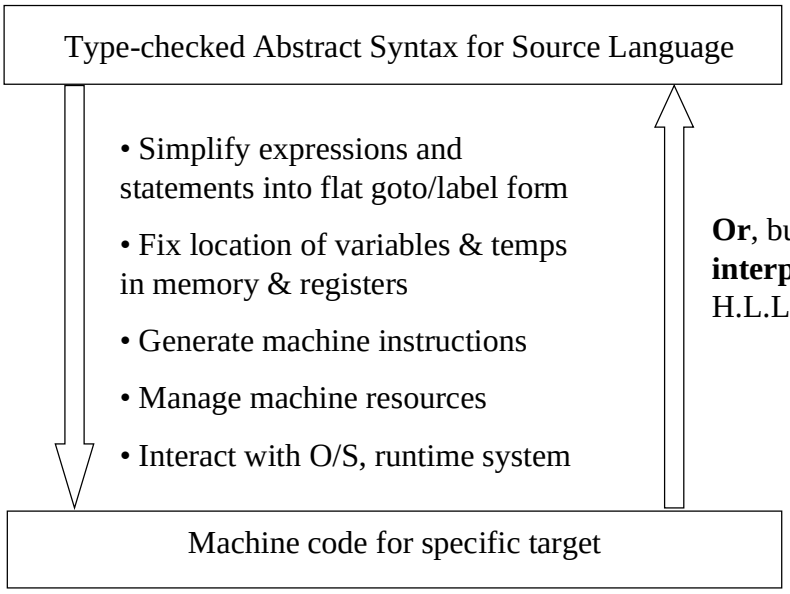
Project

- Build PCAT Interpreter
- Complete PCAT Compiler

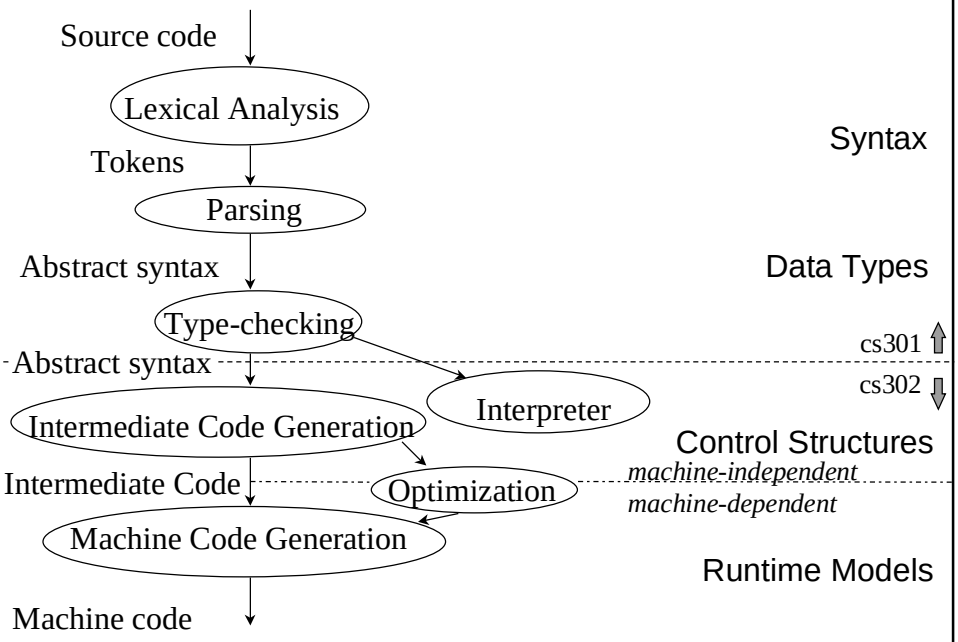
Themes

- Mapping from high-level to low-level
- Implementing resource management
- Integration with OS and hardware environment
- Syntax-directed techniques

Compiler Backend Tasks



Translator Structure



Language Design Issues

Syntax

Data Types

cs301

cs302

Control Structures

machine-independent

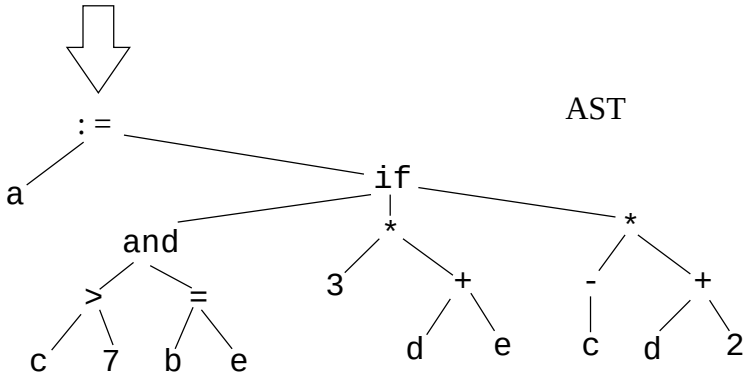
machine-dependent

Runtime Models

Translation to Abstract Syntax

```
a := if c > 7 and b = e
     then 3 * (d + e)
     else -c * (d + 2)
```

Source code



AST

Intermediate Code Generation

Emit I.C. (or "I.R.") from abstract syntax or directly from parser

Advantages:

- Keeps more of compiler machine-independent
- Facilitates some optimizations

Typical examples:

- Postfix
- Trees or DAGs
- Three-address code (quadruples, triples, etc.)

Abstracts key features of machine architectures

- E.g., sequential execution, explicit jumps

But hides details

- E.g., # of registers, style of conditionals, etc.

Many possible levels

"Three-address code" - a typical I.R.

Generate list of "instructions"

Each has an operator, up to 2 args, and up to 1 result

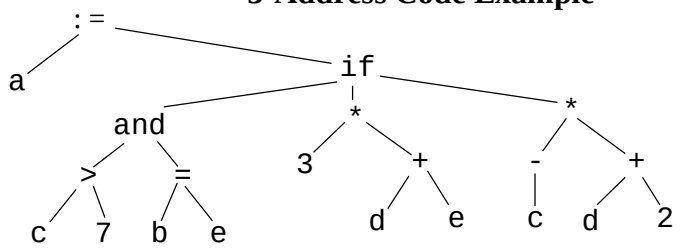
Instructions can be **labeled**

Operands are names for **locations** in some abstract memory (e.g., symbol table entries)

Examples of instructions:

```
A := B        copy
A = B op C    binary ops
A = op B      unary ops
goto L        jumps
if A relop B goto L    conditional jumps
param A       procedure call setup
call P,N      procedure call
return N      procedure return
A[I]          array dereference
```


3-Address Code Example



- Linearized
- Nested conditionals expanded (badly)
- Temporaries for all intermediate results

```
     if c > 7 goto L1
     goto L2
L1:  if b = e goto L3
L2:  t1 := d + 2
     t2 := -c
     t3 := t1 * t2
     goto L4
L3:  t1 := d + e
     t3 := 3 * t1
L4:  a := t3
```

Machine-code Generation

"Read" I.R.; generate assembly language (symbol or binary).

Must cooperate with I.R. to define and "enforce" runtime environment.

Must deal with idiosyncrasies of target machine,

- e.g., instruction selection

and perform resource management,

- e.g., register assignment.

Lots of case analysis, especially for complex target architectures.

Can do by hand, but hard.

Tools limited but sometimes useful; mainly based on pattern matching

Sample machine code

Assumes a global; b,c args; d,e locals.

Illustrates register conventions, delay slots, etc.

```
        sethi %hi(_a),%o2
        cmp %i1,7
        ble L2
        or %o2,%lo(_a),%l1
        cmp %i0,%l2
        bne L4
        sub %g0,%i1,%o0
        add %l0,%i0,%o1
```

```
        sll %o1,1,%o0
        add %o0,%o1,%o0
        b L3
        st %o0,[%o2+%lo(_a)]
L2:
        sub %g0,%i1,%o0
L4:
        call .umul,0
        add %l0,2,%o1
        st %o0,[%l1]
L3:
```

"Optimization"

Improve (**not** perfect) code by removing inefficiencies:

- In original program
- Introduced by compiler itself

Can operate on source, I.R., object code.

Local Improvements

- Example: changing

```
        if c > 7 goto L1
        goto L2
   L1: ...
   L2: ...
```

- to

```
        if c <= 7 goto L2
   L1: ...
   L2: …
```


"Optimization" (continued)

"Global" Improvements

- Example: changing

```
   for (i := 0; i < 1000; i++)
      a[i] := b*c + i;
```

- to

```
   t1 := b * c;
   for (i = 0; i < 1000; i++)
      a[i] = t1 + i;
```

Interprocedural improvements

- Example: Inlining a function

Most of a modern compiler is devoted to optimization.

Interpretation

Simulate execution of program (source, AST, or other IR) on an **abstract machine**.

Implement abstract machine on a **real** machine.

Inputs to interpreter are

- Program to be interpreted
- Input to that program

Simpler than compiling and takes no time up front, but interpreted code runs (~10X) more slowly than compiled code.

Much more portable than real machine code (as for **Java**).

Helps with semantic definition.