CSE 401: Introduction to Compiler Construction

Professor: Craig Chambers
TAs: Vass Litvinov, Zasha Weinberg

Text: *Compilers: Principles, Techniques, and Tools*, Aho *et al.*

Goals:
- learn principles & practice of language implementation
  - brings together theory & pragmatics of previous courses
- study interactions among:
  - language features
  - implementation efficiency
  - compiler complexity
  - architectural features
- gain experience with object-oriented design & C++
- gain experience working on a team

Prerequisites:
- 326, 341, 378
- very helpful: 322

Course Outline

Front-end issues:
- lexical analysis (scanning): characters → tokens
- syntax analysis (parsing): tokens → abstract syntax trees
- semantic analysis (typechecking): annotate ASTs

Midterm

Back-end issues:
- run-time storage representations
- intermediate & target code generation: ASTs → asm code
- optimizations

Final

Project

Start with compiler for PL/0, written in C++

Add:
- comments
- arrays
- call-by-reference arguments
- results of procedures
- for loops
- break statements
- and more...

Completed in stages over the quarter

Strongly encourage working in a 2-person team on project

Grading based on:
- correctness
- clarity of design & implementation
- quality of test cases

Grading

Project: 40% total
Homework: 20% total
Midterm: 15%
Final: 25%

Homework & projects due at the **start of class**

3 free late days, per person
- thereafter, 25% off per calendar day late

An example compilation

Sample PL/0 program: `squares.0`

```
module main;
   var x:int, result:int;
   procedure square(n:int);
   begin
      result := n * n;
   end square;
begin
   x := input;
   while x <> 0 do
      square(x);
      output := result;
      x := input;
   end;
end main.
```


First step: lexical analysis

“Scanning”, “tokenizing”

Read in characters, clump into **tokens**

- strip out whitespace in the process

Specifying tokens: regular expressions

Example:

```
Ident    ::= Letter AlphaNum*
Integer  ::= Digit+
AlphaNum ::= Letter | Digit
Letter   ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
Digit    ::= '0' | ... | '9'
```


Second step: syntax analysis

“Parsing”

Read in tokens, turn into a tree based on syntactic structure

Specifying syntax: context-free grammars

EBNF is a popular notation for CFG's

Example:

```
Stmt      ::= AsgnStmt | IfStmt | ...
AsgnStmt  ::= LValue := Expr ;
LValue    ::= Id
IfStmt    ::= if Test then Stmt [else Stmt] ;
Test      ::= Expr = Expr | Expr < Expr | ...
Expr      ::= Term + Term | Term - Term | Term
Term      ::= Factor * Factor | ... | Factor
Factor    ::= - Factor | Id | Int | ( Expr )
```

EBNF specifies *concrete syntax* of language

Parser usually constructs tree representing *abstract syntax* of language

Third step: semantic analysis

"Name resolution and typechecking"

Given AST:

- figure out what declaration each name refers to
- perform static consistency checks

Key data structure: symbol table

- maps names to info about name derived from declaration

Semantic analysis steps:

1. Process each scope, top down
2. Process declarations in each scope into symbol table for scope
3. Process body of each scope in context of symbol table

Fourth step: storage layout

Given symbol tables,
determine how & where variables will be stored at run-time

What representation for each kind of data?

How much space does each variable require?

In what kind of memory should it be placed?

- static, global memory
- stack memory
- heap memory

Where in that kind of memory should it be placed?

- e.g. what stack offset

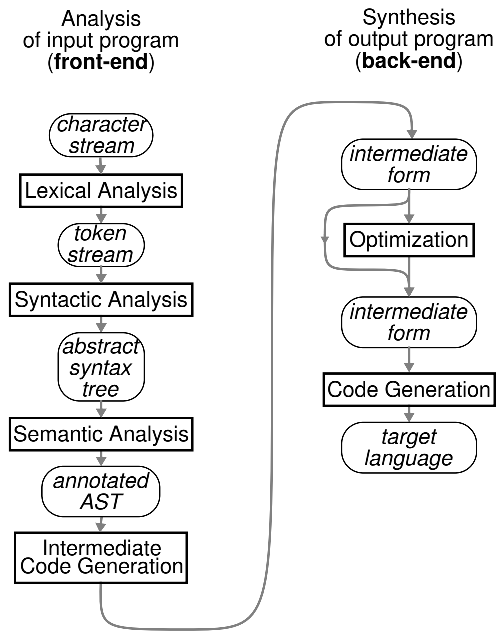
Fifth step: intermediate & target code generation

Given annotated AST & symbol tables,
produce target code

Often done as three steps:

- produce machine-independent low-level representation of program (intermediate representation)
- perform machine-independent optimizations of IR (optional)
- translate IR into machine-specific target instructions
  - instruction selection
  - register allocation

Summary of compiler phases



The bigger picture

Compilers are *translators*

Characterized by

- input language
- target language
- degree of “understanding”

Compilers vs. interpreters

Compilers implement languages by translation

Interpreters implement languages directly

Embody different trade-offs among:

- execution speed of program
- start-up overhead, turn-around time
- ease of implementation
- programming environment facilities
- conceptual difficulty

Engineering issues

Portability

- ideal: multiple front-ends & back-ends sharing intermediate language

Sequencing phases of compilation

- stream-based
- syntax-directed

Multiple, separate passes vs. fewer, integrated passes

How to avoid compiler bugs?