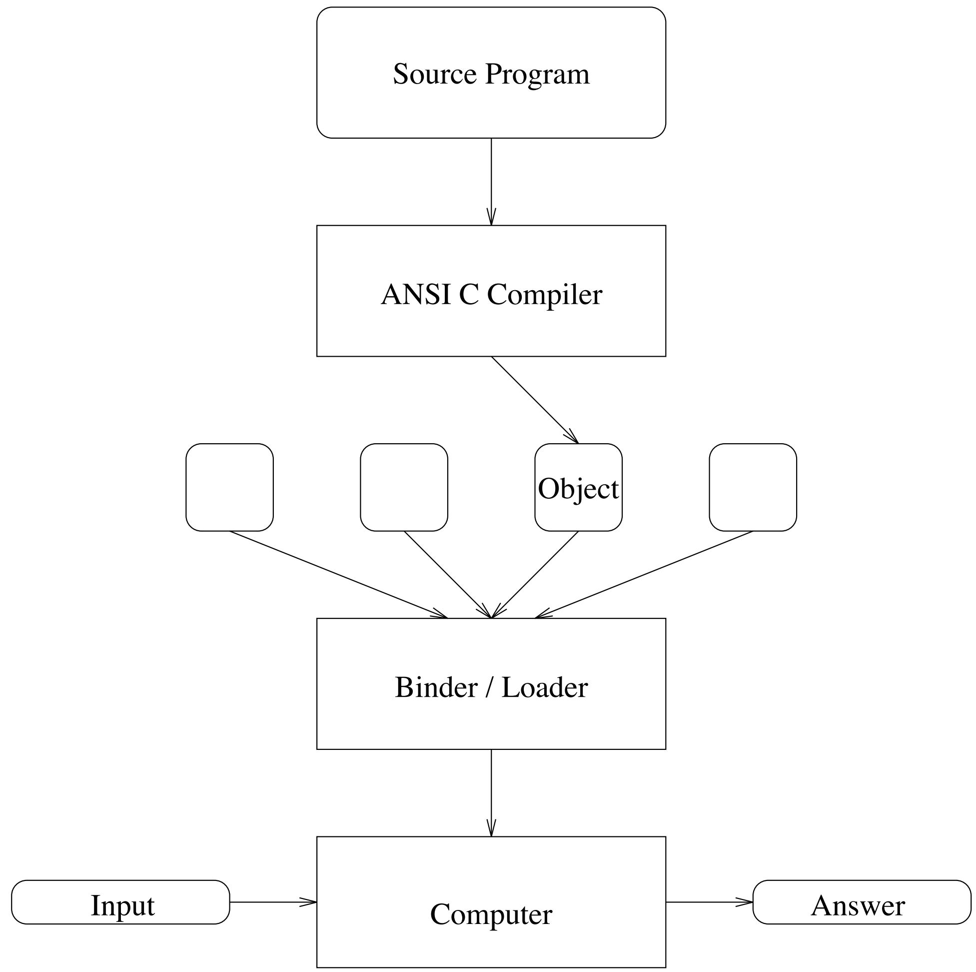
Tutorial outline

1. Introduction
2. Scanning and tokenizing
3. Grammars and ambiguity
4. Recursive descent parsing
5. Error repair
6. Table-driven LL parsing
7. Bottom-up table-driven parsing
8. Symbol tables
9. Semantic analysis via attributes
10. Abstract syntax trees
11. Type checking
12. Runtime storage management
13. Code generation
14. Optimization
15. Conclusion

What is a language translator?

You type: cc foo.c. . . What happens?



Language: Vehicle (architecture) for transmitting information between components of a system. For our purposes, a language is a *formal interface*. The goal of every compiler is correct and efficient language translation.

The process of language translation

1. **A person has an idea of how to compute something:**

$$fact(n) = \begin{cases} 1 & \textbf{if } n \le 0 \\ n \times fact(n-1) & \textbf{otherwise} \end{cases}$$

2. **An algorithm captures the essence of the computation:**

$$fact(n) = \textbf{if } n \le 0 \textbf{ then } 1 \textbf{ else } n \times fact(n-1)$$

Typically, a *pseudocode* **language is used, such as "pidgin ALGOL".**

3. **The algorithm is expressed in some programming language:**

```
int fact(int n) {
   if (n <= 0) return(1);
   else return(n*fact(n-1));
}
```

We would be done if we had a computer that "understood" the language directly. So why don't we build more C machines?

a) **How does the machine know it's seen a** C **program and not a Shakespeare sonnet?**

b) **How does the machine know what is "meant" by the** C **program?**

c) **It's hard to build such machines. What happens when language extensions are introduced (**C++**)?**

d) **RISC philosophy says simple machines are better.**

Finally...

A compiler translates programs written in a *source* language into a *target* language. For our purposes, the source language is typically a *programming language*—convenient for humans to use and understand—while the target language is typically the (relatively low-level) instruction set of a computer.

Source Program

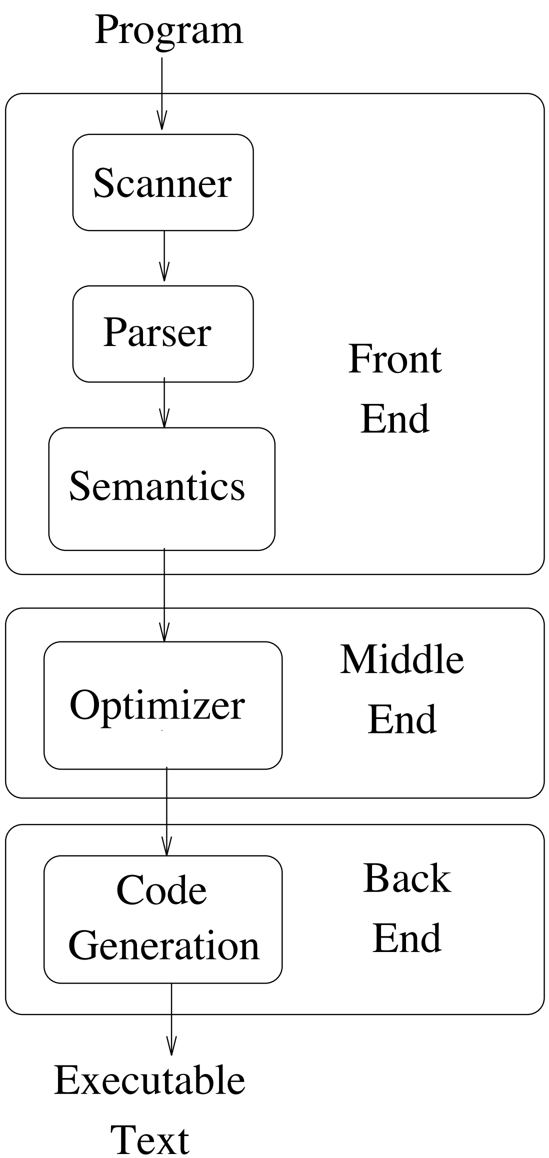
```
main() {
   int a;

   a += 5.0;
}
```

Target Program (Assembly)

```
_main:
    !#PROLOGUE# 0
    sethi   %hi(LF12),%g1
    add %g1,%lo(LF12),%g1
    save    %sp,%g1,%sp
    !#PROLOGUE# 1
    sethi   %hi(L2000000),%o0
    ldd [%o0+%lo(L2000000)],%f0
    ld  [%fp+-0x4],%f2
    fitod   %f2,%f4
    faddd   %f4,%f0,%f6
    fdtoi   %f6,%f7
    st  %f7,[%fp+-0x4]
```

Running the Sun `cc` compiler on the above source program of 32 characters produces the assembly program shown to the right. The bound binary executable occupied in excess of 24 thousand bytes.

Structure of a compiler

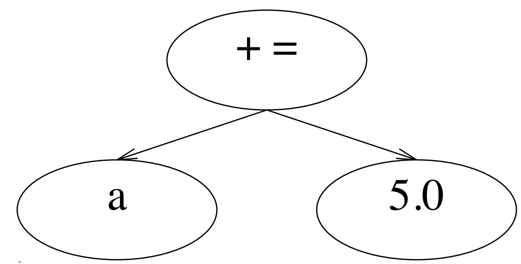


Front End

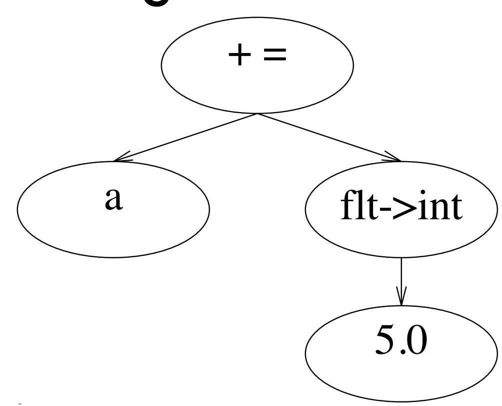
Scanner: decomposes the input stream into *tokens*. So the string "a += 5.0;" becomes

$\boxed{\mathbf{a}}\ \boxed{+=}\ \boxed{5.0}\ \boxed{;}$

Parser: analyzes the tokens for correctness and structure:

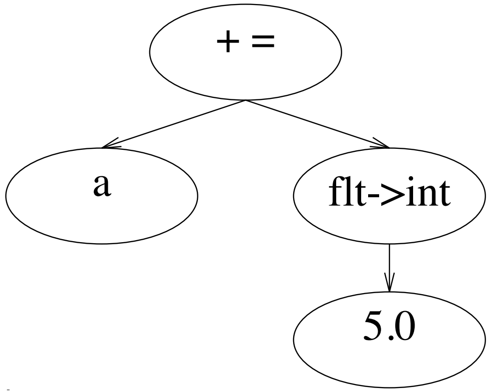


Semantic analysis: more analysis and type checking:

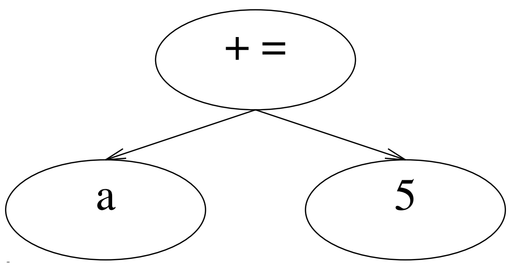


Structure of a compiler

Middle End



The middle end might eliminate the conversion, substituting the integer "5" for the float "5.0".



Code Generation

The code generator can significantly affect performance. There are many ways to compute "a+=5", some less efficient than others:

while $(t \neq a + 5)$ **do**
$t \leftarrow rand()$
od
$a \leftarrow t$

While optimization can occur throughout the translation process, machine-independent transformations are typically relegated to the middle-end, while instruction selection and other machine-specific activities are pushed into code generation.

Bootstrapping a compiler

Often, a compiler is written in it "itself". That is, a compiler for PASCAL **may be written in** PASCAL**. How does this work?**

Initial Compiler for L on Machine M

1. **The compiler can be written in a small subset of L, even though the compiler translates the full language.**
2. **A throw-away version of the subset language is implemented on M. Call this compiler α.**
3. **The L compiler can be compiled using the subset compiler, to generate a full compiler β.**
4. **The L compiler can also compile itself. The resulting object γ can be compared with β for verification.**

Porting the Compiler

1. **On machine M, the code generator for the full compiler is changed to target machine N.**
2. **Any program in L can now be cross-compiled from M to N.**
3. **The compiler can also be cross-compiled to produce an instance of γ that runs on machine N.**

If the run-time library is mostly written in L, or in an intermediate language of β, then these can also be translated for N using the cross-compiler.

What else does a compiler do?

```
if (p)
   a = b + (c
else {d = f;
q = r;
```

Error detection. Strict language rules, consistently enforced by a compiler, increase the likelihood that a compiler-approved source program is bug-free.

Error diagnosis. Compilers can often assist the program author in addressing errors.

Error repair. Some ambitious compilers go so far as to insert or delete text to render the program executable.

```
for (i=1; i<=n; ++i)
{
    a[i] = b[i] + c[i]
}
```

Program optimization. The target produced by a compiler must be "observably equivalent" to the source interpretation. An optimizing compiler attempts to minimize the resource constraints (typically time and space) required by the target program.

Program instrumentation. The target program can be augmented with instructions and data to provide information for run-time debugging and performance analysis. Language features not checkable at compile-time are often checked at run-time by code inserted by the compiler.

Sophisticated error repair may include symbol insertion, deletion, and use of indentation structure.

Program optimization can significantly decrease the time spent on array index arithmetic. Since subscript ranges cannot in general be checked at compile-time, run-time tests may be inserted by the compiler.

Compiler design points – aquatic analogies

Powerboat Turbo–?. These compilers are fast, load-and-go. They perform little optimization, but typically offer good diagnostics and a good programming environment (sporting a good debugger). These compilers are well-suited for small development tasks, including small student projects.

Sailboat BCPL, Postscript. These compilers can do neat tricks but they require skill in their use. The compilers themselves are often small and simple, and therefore easily ported. They can assist in bootstrapping larger systems.

Tugboat C++ preprocessor, RATFOR. These compilers are actually front-ends for other (typically larger) back-ends. The early implementations of C++ were via a preprocessor.

Barge Industrial-strength. These compilers are developed and maintained with a company's reputation on the line. Commercial systems use these compilers because of their integrity and the commitment of their sponsoring companies to address problems. Increasingly these kinds of compilers are built by specialty houses such as Rational, KAI, etc.

Ferry Gnu compilers. These compilers are available via a General Public License from the Free Software Foundation. They are high-quality systems and can be built upon without restriction.

Another important design issue is the extent to which a compiler can respond *incrementally* to changes.

Compilers are taking over the world!

While compilers most prevalently participate in the translation of programming languages, some form of compiler technology appears in many systems:

Text processing Consider the "$\star$-roff" text processing pipe:

$$\text{PIC} \rightarrow \text{TBL} \rightarrow \text{EQN} \rightarrow \text{TROFF}$$

or the LaTeX pipe:

$$\text{\LaTeX} \rightarrow \text{\TeX}$$

each of which may produce

$$\text{DVI} \rightarrow \text{POSTSCRIPT}$$

Silicon compilers Such systems accept circuit specifications and compile these into VLSI layouts. The compilers can enforce the appropriate "rules" for valid circuit design, and circuit libraries can be referenced like modules in software library.

Compiler design *vs.* programming language design

Programming languages have	So compilers offer
Non-locals	Displays, static links
Recursion	Dynamic links
Dynamic Storage	Garbage collection
Call-by-name	Thunks
Modular structure	Interprocedural analysis
Dynamic typing	Static type analysis

It's expensive for a compiler to offer	So some languages avoid that feature
Non-locals	C
Call-by-name	C, PASCAL
Recursion	FORTRAN 66
Garbage collection	C

In general, *simple* languages such as C, PASCAL, and SCHEME have been more successful than complicated languages like PL/1 and ADA.

Language design for humans

Procedure $foo(x, y)$

declare

x, y **integer**
a, b **integer**
$\star p$ **integer**

$p \leftarrow rand()$? &a : &b

$\star p \leftarrow x + y$

end

Syntactic simplicity. Syntactic signposts are kept to a minimum, except where aesthetics dictate otherwise: parentheses in C, semicolons in PASCAL.

Resemblance to mathematics. Infix notation, function names.

Flexible internal structures. Nobody would use a language in which one had to predeclare how many variables their program needed.

Freedom from specifying side-effects. What happens when p is dereferenced?

Programming language design is often a compromise between ease of use for humans, efficiency of translation, and efficiency of target code execution.

Language design for machines

```
(SymbolTable
   (NumSymbols 5)
   (Symbol
      (SymbolName x)
      (SymbolID   1)
   )
   (Symbol
      (SymbolName y)
      (SymbolID   2)
   )
   ...
)
(AliasRelations
   (NumAliasRelations 1)
   (AliasRelation
      (AliasID 1)
      (MayAliases 2 a b)
   )
)
```

```
(NodeSemantics
  (NodeID 2)
  (Def
    (DefID 2)
    (SymbID ?)
    (AliasWith 1)
    (DefValue
      (+
        (Use
          (UseID 1)
          (SymbID x)
        )
        (Use
          (UseID 2)
          (SymbID y)
        )
      )
    )
  )
)
```

We can require much more of our intermediate languages, in terms of details and syntactic form.

Compilers and target instruction sets

How should we translate $X = Y + Z$

In the course of its code generation, a simple compiler may use only 20% of a machine's potential instructions, because anomalies in an instruction set are difficult to "fit" into a code generator.

Consider two instructions

ADDREG	R_1 R_2	$R_1 \leftarrow R_1 + R_2$
ADDMEM	R_1 Loc	$R_1 \leftarrow R_1 + \star Loc$

Each instruction is *destructive* in its first argument, so Y and Z would have to be refetched if needed.

LOAD	**1**	**Y**
ADDMEM	**1**	**Z**
STORE	**1**	**X**

A simpler model would be to do all arithmetic in registers, assuming a *nondestructive* instruction set, with a reserved register for results (say, R_0):

LOAD	**1**	**Y**
LOAD	**2**	**Z**
LOADREG	**0**	**1**
ADDREG	**0**	**2**
STORE	**0**	**X**

This code preserves the value of Y and Z in their respective registers.

A popular approach is to generate code assuming the nondestructive paradigm, and then use an instruction selector to optimize the code, perhaps using destructive operations.

Current wisdom on compilers and architecture

Architects should design "orthogonal" RISC instruction sets, and let the optimizer make the best possible use of these instructions. Consider the program

$$\textbf{for } i \leftarrow 1 \textbf{ to } 10 \textbf{ do } X \leftarrow A[i]$$

where A is declared as a 10-element array ($1 \ldots 10$).

The VAX **has an instruction essentially of the form**

$$Index(A, i, low, high)$$

with semantics

> **if** $(low \leq i \leq high)$ **then**
>   **return** $(A + 4 \times i)$
> **else**
>   **return** $(error)$
> **fi**

Internally, this instruction requires two tests, one multiplication, and one addition.

However, notice that the loop does not violate the array bounds of A. Moreover, in moving from $A[i]$ to $A[i+1]$, the new address can be calculated by adding 4 to the old address.

While the use of an $Index$ instruction may seem attractive, better performance can be obtained by providing smaller, faster instructions to a compiler capable of optimizing their use.