

# CMPT 755

## Compilers

Anoop Sarkar

<http://www.cs.sfu.ca/~anoop>

# Compilers

- Analysis of the source (front-end)
- Synthesis of the target (back-end)
- The *translation* from user **intention** into intended **meaning**
- The requirements from a Compiler and a Programming Language are:
  - Ease of use (high-level programming)
  - Speed

# Cousins of the compiler

- “Smart” editors for structured languages
  - static checkers; pretty printers
- Structured or semi-structured data
  - Trees as data: s-expressions; XML
  - query languages for databases: SQL
- Interpreters (for PLs like lisp or scheme)
  - Scripting languages: perl, python, tcl/tk
  - Special scripting languages for applications
  - “Little” languages: awk, eqn, troff, TeX
- Compiling to Bytecode (virtual machines)

# Context for the Compiler

- Preprocessor
- Compiler
- Assembler
- Linker
  - a.out, exec & link format: ELF, common  
object file format: COFF
- Loader

# What we understand

```
#include <stdio.h>
```

```
int main (int argc, char *argv[]) {  
    int i;  
    int sum = 0;  
    for (i = 0; i <= 100; i++)  
        sum = sum + i * i;  
    printf ("Sum from 0..100 = %d\n", sum);  
}
```

```
001001111011110111111111111111100000
1010111110111111000000000000010100
101011111010010000000000000100000
101011111010010100000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
001001011100100000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
000000000000000000111100000010010
00000011000011111100100000100001
000101000010000011111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
000011000001000000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
0000001111100000000000000001000
000000000000000000001000000100001
```

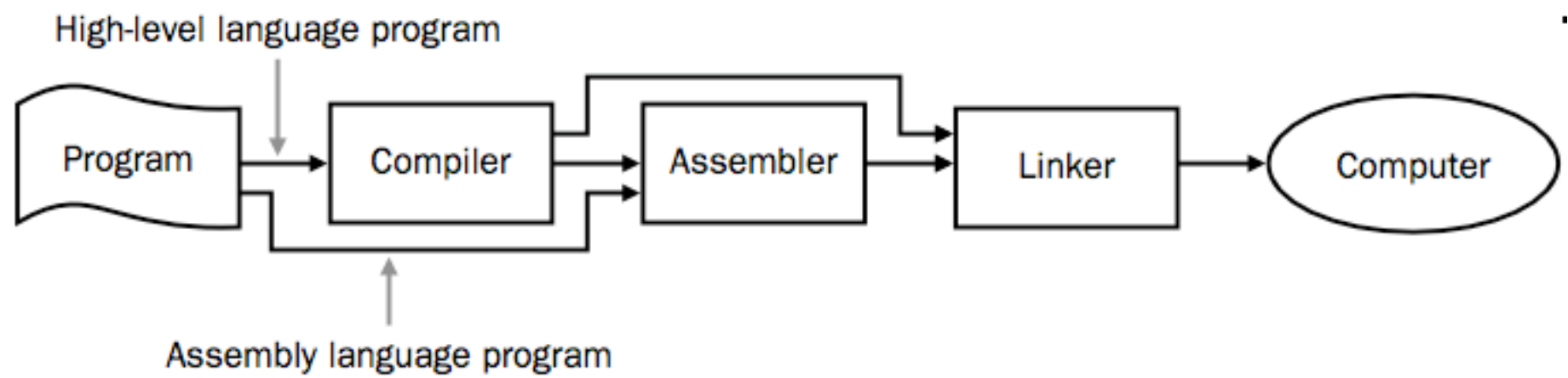
Conversion into  
instructions for  
the Machine

MIPS  
machine language  
code

# Assembly language

```
.text
        .globl main
main:
        ori $8, $0, 2
        ori $9, $0, 3
        addu $10, $8, $9
```

A one-one translation from machine code to assembly  
(assuming a single file of assembly with no dependencies)





# Linker

.data

str:

.asciiz "the answer = "

.text

main:

li \$v0, 4

la \$a0, str

syscall

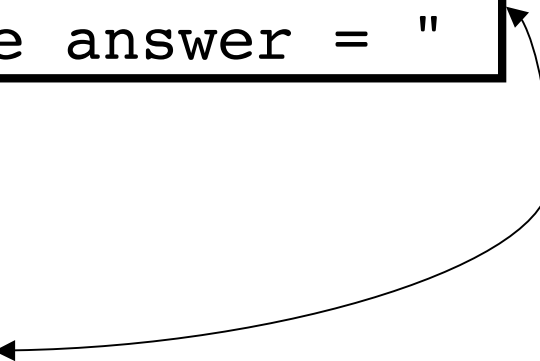
li \$v0, 1

li \$a0, 42

syscall

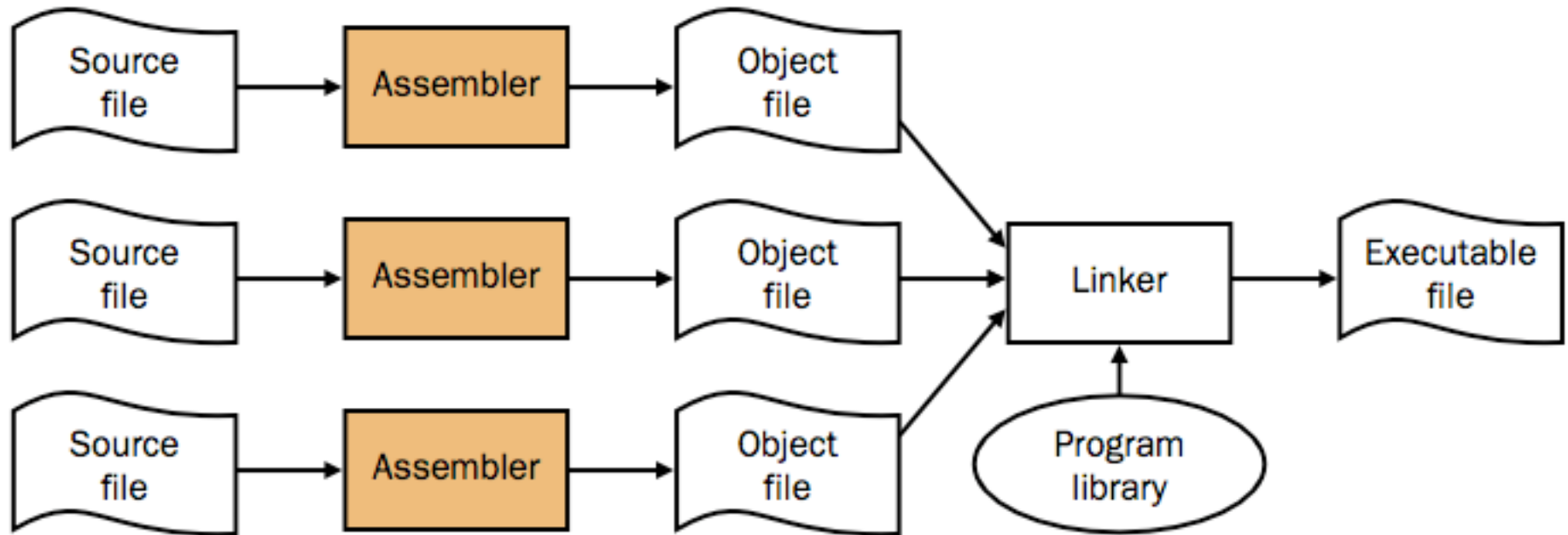
Local vs. Global labels

2-pass assembler and Linker



# The UNIX toolchain

(as, ar, ranlib, libtool, ld, ...)



# Historical Background

- Machine language/Assembly language
- 1957: First FORTRAN compiler
  - 18 man years of effort
- Today's techniques were created in response to the difficulties of implementing early compilers

# Programming Language Design

- Ease of use (difficult: depends on the zeitgeist)
- Simplicity
- Visualize the dynamic process of the programs runtime by examining the static program code
- Code reuse: polymorphic functions, objects
- Checking for correctness: strong vs. weak typing, side-effects, formal models
- The less typing the better: syntactic “sugar”
- Automatic memory management
- Community acceptance: extensions and libraries

# Programming Language Design

- Speed (closely linked to the compiler tools)
- Defining tokens
- Defining the syntax
- Defining the “semantics” (typing, polymorphism, coercion, etc.)
- Core language vs. the standard library
- Hooks for code optimization (iterative idioms vs. pure functional languages)

# Building a compiler

- The cost of compiling and executing should be managed
- No program that violates the definition of the language should escape
- No program that is valid should be rejected

# Building a compiler

- Requirements for building a compiler:
  - Symbol-table management
  - Error detection and reporting
- Stages of a compiler:
  - Analysis (front-end)
  - Synthesis (back-end)

# Stages of a Compiler

- Analysis (Front-end)
  - Lexical analysis
  - Syntax analysis (parsing)
  - Semantic analysis (type-checking)
- Synthesis (Back-end)
  - Intermediate code generation
  - Code optimization
  - Code generation



# Lexical Analysis

- Also called *scanning*, take input program *string* and convert into tokens
- Example:

```
double f = sqrt(-1);
```

T_DOUBLE	("double")
T_IDENT	("f")
T_OP	("=")
T_IDENT	("sqrt")
T_LPAREN	(" (")
T_OP	(" -")
T_INTCONSTANT	("1")
T_RPAREN	(" )")
T_SEP	(" ;")

# Syntax Analysis

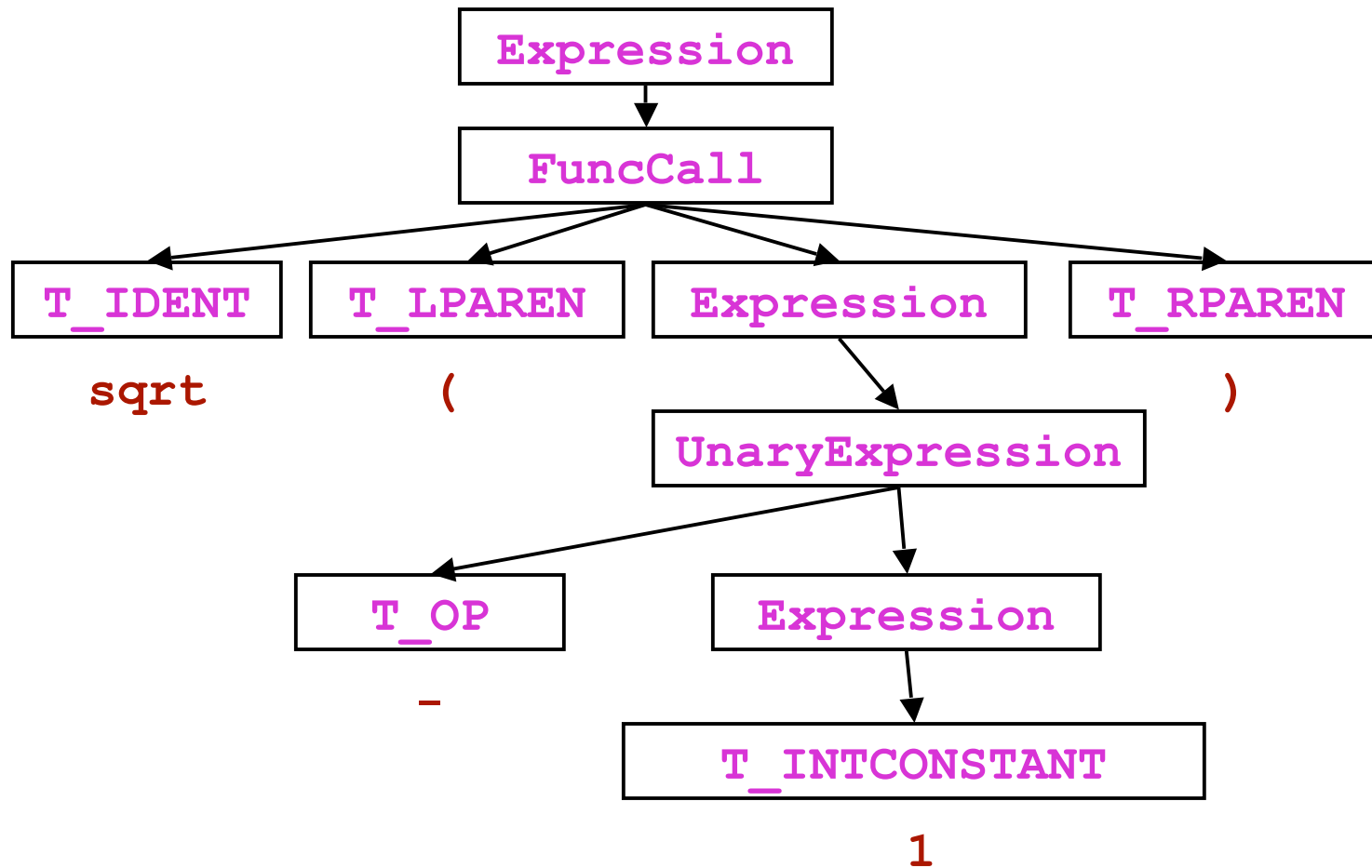
- Also called *parsing*
- Describe the set of strings that are programs using a grammar
- Pick the simplest grammar formalism possible (but not too simple)
  - Finite-state machines (Regular grammars)
  - Deterministic Context-free grammars
  - Context-free grammars
- Structural validation
- Creates parse tree or derivation

# Derivation of `sqrt(-1)`

```
Expression -> UnaryExpression
Expression -> FuncCall
Expression -> T_INTCONSTANT
UnaryExpression -> T_OP Expression
FuncCall -> T_IDENT T_LPAREN Expression T_RPAREN
```

```
Expression
-> FuncCall
-> T_IDENT T_LPAREN Expression T_RPAREN
-> T_IDENT T_LPAREN UnaryExpression T_RPAREN
-> T_IDENT T_LPAREN T_OP Expression T_RPAREN
-> T_IDENT T_LPAREN T_OP T_INTCONSTANT T_RPAREN
```

# Parse Trees



# Semantic analysis

- “does it make sense”?
- Checking semantic rules, such as
  - Is there a `main` function?
  - Is variable declared?
  - Are operand types compatible? (coercion)
  - Do function arguments match function declarations?
- Static vs. run-time semantic checks
  - Array bounds, return values do not match definition

# Intermediate Code Generation

- Three-address code (TAC)

```
j = 2 * i + 1;  
if (j >= n)  
    j = 2 * i + 3;  
return a[j];
```

```
    _t1 = 2 * i  
    _t2 = _t1 + 1  
    j = _t2  
    _t3 = j < n  
    if _t3 goto L0  
    _t4 = 2 * i  
    _t5 = _t4 + 3  
    j = _t5  
L0:  _t6 = a[j]  
    return _t6
```

# Code Optimization

- Example

```
    _t1 = 2 * i
    _t2 = _t1 + 1
    j = _t2
    _t3 = j < n
    if _t3 goto L0
    _t4 = 2 * i
    _t5 = _t4 + 3
    j = _t5
L0:  _t6 = a[j]
    return _t6
```

```
    _t1 = 2 * i

    j = _t1 + 1
    _t3 = j < n
    if _t3 goto L0

    j = _t1 + 3

L0:  _t6 = a[j]
    return _t6
```

# Object code generation

- Example:  $a$  in  $\$a0$ ,  $i$  in  $\$a1$ ,  $n$  in  $\$a2$

```
_t1 = 2 * i
```

```
j = _t1 + 1
```

```
_t3 = j < n
```

```
if _t3 goto L0
```

```
j = _t1 + 3
```

```
mulo $t1, $a0, 2
```

```
add $s0, $t1, 2
```

```
seq $t2, $s0, $a2
```

```
beq $t2, 1, L0
```

```
add $s0, $t1, 3
```



# Bootstrapping a Compiler

- Machine code at the beginning
- Make a simple subset of the language, write a compiler for it, and then use that subset for the rest of the language definition
- Bootstrap from a simpler language
  - C++ (“C with classes”)
- Interpreters
- Cross compilation

# Wrap Up

- Analysis/Synthesis
  - Translation from string to executable
- Divide and conquer
  - Build one component at a time
  - Theoretical analysis will ensure we keep things **simple** and **correct**
  - Create a complex piece of software