CS451 – Software Analysis

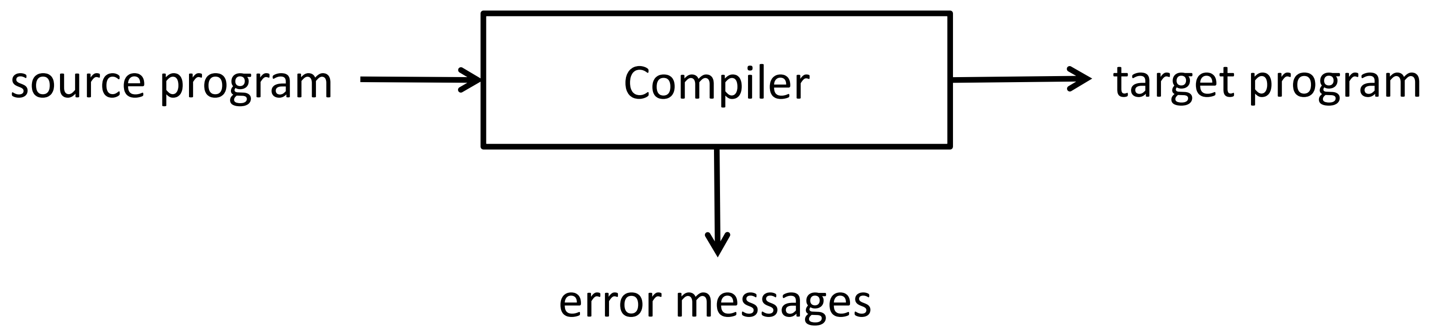
Lecture 17

Introduction to Compilers

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy

What is a compiler?

- A compiler is a program that
  - reads a program written in one language (source)
  - and translates it to an equivalent program in another language (target)
  - **important**: error reporting during translation



Examples

- GCC (Gnu Compiler Collection)
  - gcc, g++, javac, etc.
- LLVM (Low Level Virtual Machine)
  - clang, clang++
- “Compilers” are everywhere,
  - Pretty printers for colored syntax in editors, static checkers, interpreters for scripting languges, etc.

Analysis-synthesis model

- There are two parts in compilation:
  - Analysis
  - Synthesis
- Analysis
  - Breaks up the *source program* to subparts and creates intermediate representation(s)
- Synthesis
  - Constructs the *target program* from intermediate representation(s)

Example

```
\begin{table}[tb]
    \centering
    \caption{We name gadgets based on their type (prefix), payload (body),
    and exit instruction (suffix). In total, we name 2$\times$3$\times$3=18
    different gadget types.}
    \begin{tabular}{|c|c|c|}
        \hline
        \textbf{Gadget type} & \textbf{Payload instructions} &
            \textbf{Exit instruction} \\
        \hline
        {Prefix} & {Body} & {Suffix} \\
        \hline

        \begin{tabular}{l}
        CS - Call site\\
        EP - Entry point\\
        \end{tabular} &
        ...
```

```
$ pdflatex main.tex
```

TABLE II: We name gadgets based on their type (prefix), payload (body), and exit instruction (suffix). In total, we name $2\times3\times3=18$ different gadget types.

Gadget type	**Payload instructions**	**Exit instruction**
Prefix	Body	Suffix
CS - Call site EP - Entry point	IC - Indirect call F - Fixed function call *none* - Other instructions	R - Return IC - Indirect call IJ - Indirect jump

Requirements

- Compiler
  - Reliability
  - Fast execution
  - Low memory overhead
  - Good error reporting
  - Error recovery
  - Portability
  - Maintainability
- Target program
  - Fast execution
  - Low memory overhead

Source code

- Easy to read/write by human

```
int expr(int n) {
    int d;

    d = 4 * n * n * (n + 1) * (n + 1);

    return d;
}
```

Assembly and machine code

- Optimized for execution by a machine (CPU)
- Less descriptive
- Hard to be processed by a human

```
lda $30,-32($30)
stq $26,0($30)
stq $15,8($30)
bis $30,$30,$15
bis $16,$16,$1
stl $1,16($15)
lds $f1,16($15)
sts $f1,24($15)
ldl $5,24($15)
bis $5,$5,$2
s4addq $2,0,$3
ldl $4,16($15)
mull $4,$3,$2
ldl $3,16($15)
```

Optimizations

- Compilers have several layers of optimizations

```
int expr(int n){
    int d;
    d = 4 * n * n * (n + 1) * (n + 1);
    return d;
}
```

No optimizations

$ gcc –O0

```
.expr:
    stw 31,-4(1)
    stwu 1,-40(1)
    mr 31,1
    stw 3,64(31)
    lwz 0,64(31)
    mr 9,0
    slwi 0,9,2
    lwz 9,64(31)
    mullw 0,0,9
    lwz 11,64(31)
    addi 9,11,1
    mullw 0,0,9
    lwz 11,64(31)
    addi 9,11,1
    mullw 0,0,9
    stw 0,24(31)
    lwz 0,24(31)
    mr 3,0
    b L..2
L..2:
    lwz 1,0(1)
    lwz 31,-4(1)
    blr
```

Optimizations

$ gcc –O3

```
.expr:
    addi 9,3,1
    slwi 0,3,2
    mullw 3,3,0
    mullw 3,3,9
    mullw 3,3,9
    blr
```

Cross-compiler

- Compilers can generate code for different machines (targets)

```
int expr(int n){
    int d;
    d = 4 * n * n * (n + 1) * (n + 1);
    return d;
}
```

For x86

$ gcc –O3 –b i586

```
expr:
   pushl    %ebp
   movl     %esp, %ebp
   movl     8(%ebp), %eax
   leal     1(%eax), %edx
   imull    %eax, %eax
   imull    %edx, %eax
   imull    %edx, %eax
   sall     $2, %eax
   popl     %ebp
   ret
```

For PowerPC

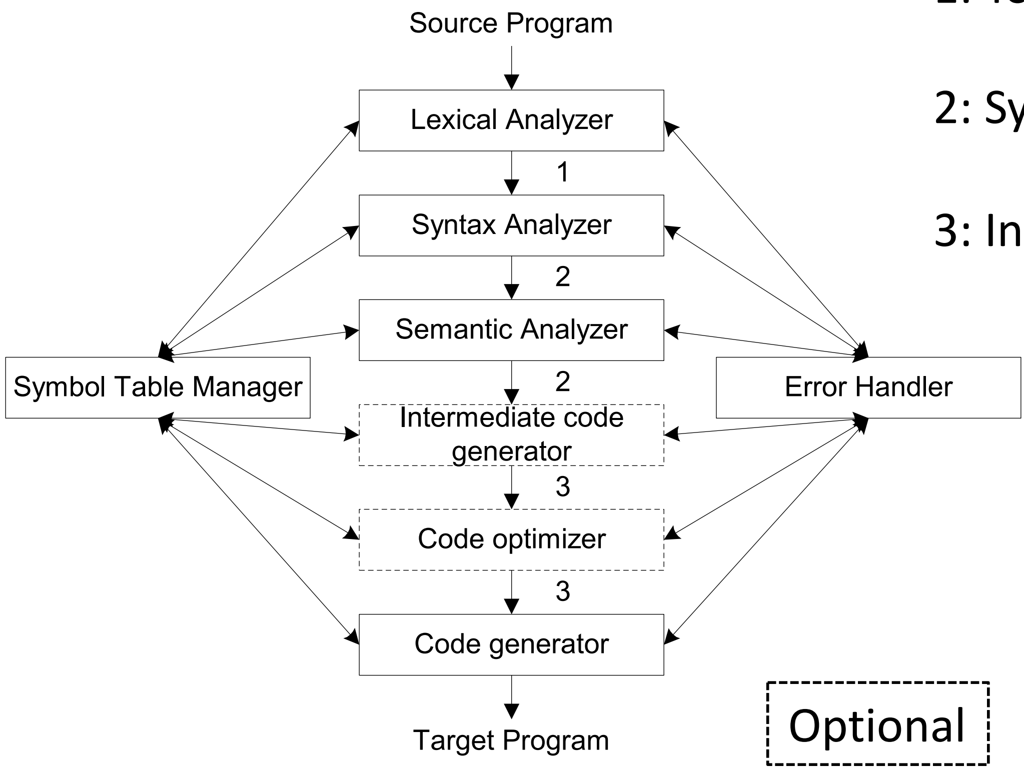
$ gcc –O3 –b powerpc

```
.expr:
    addi  9,3,1
    slwi  0,3,2
    mullw 3,3,0
    mullw 3,3,9
    mullw 3,3,9
    blr
```

Compilation life cycle

- Phases
  - Source code is transformed to intermediate representations
  - Each intermediate representation is suitable for a particular processing (lexical, syntax, optimization, etc.)
- In each phase the program is translated to a form closer to the machine representation and less similar to the (human-oriented) source representation

Compiler Phases



1: Tokens

2: Syntax tree

3: Intermediate code

Optional

Analysis of the source program

- Linear analysis
  - Source is treated as a stream of characters (left-to-right) and is grouped into tokens
- Hierarchical analysis
  - Tokens are further grouped in larger grammatical structures (e.g., nested parentheses and blocks)
- Semantic analysis
  - Certain checks are performed to ensure the validity of the identified grammatical structures

Lexical analysis

- Linear scanning
- Consider the expression

```
position := initial + rate *60
```

- Lexical analysis produces

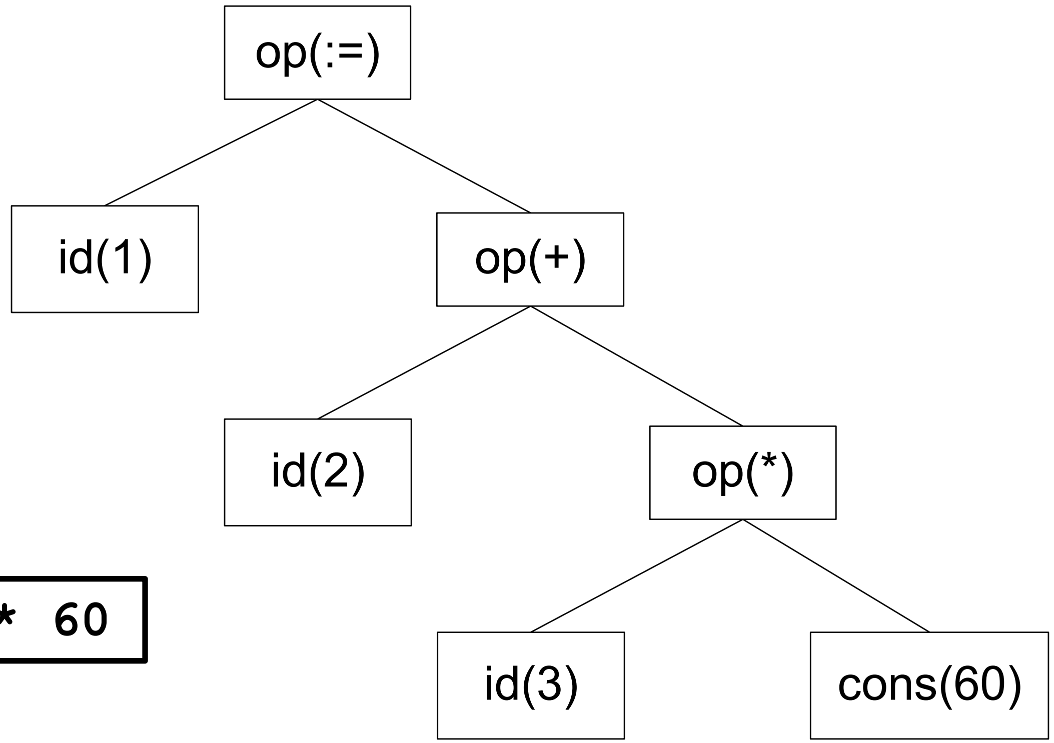
```
id(1) op(:=) id(2) op(+) id(3) op(*) cons(60)
id: identifier, op: operator, cons: constant
```

- Symbol Table

1	position	…
2	initial	…
3	rate	…
4	…	…

Syntax analysis

- Hierarchical
- Involves grouping the tokens into grammatical phases
- Constructs the structure with the token relationship




```
position := initial + rate * 60
```

Simple Grammar

- The hierarchical structure of the program is usually expressed by recursive rules

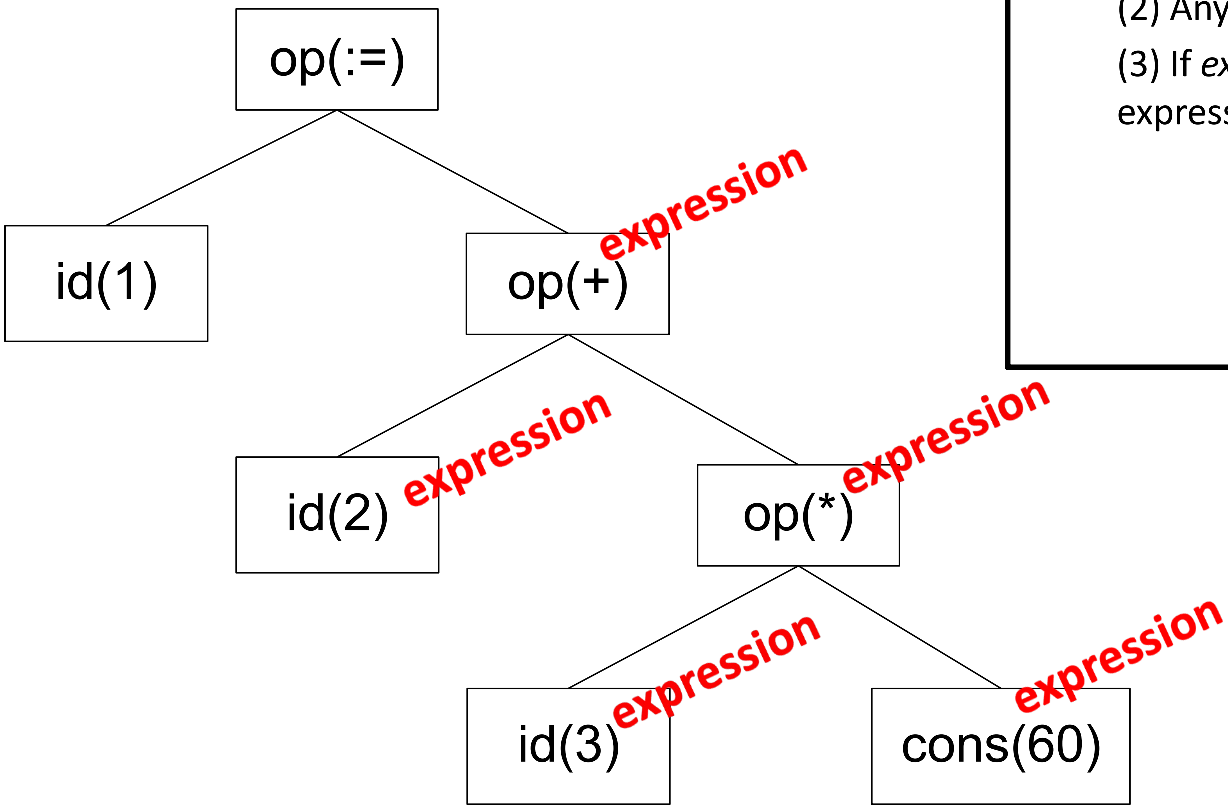
1. Any *identifier* is an expression
2. Any *number* is an expression
3. If $expression_1$ and $expression_2$ are expressions, then so are:

   $expression_1 + expression_2$

   $expression_1 * expression_2$

   $(expression_1)$

Applying the grammar



(1) Any *identifier* is an expression

(2) Any *number* is an expression

(3) If $expression_1$ and $expression_2$ are expressions, then so are:

$expression_1 + expression_2$

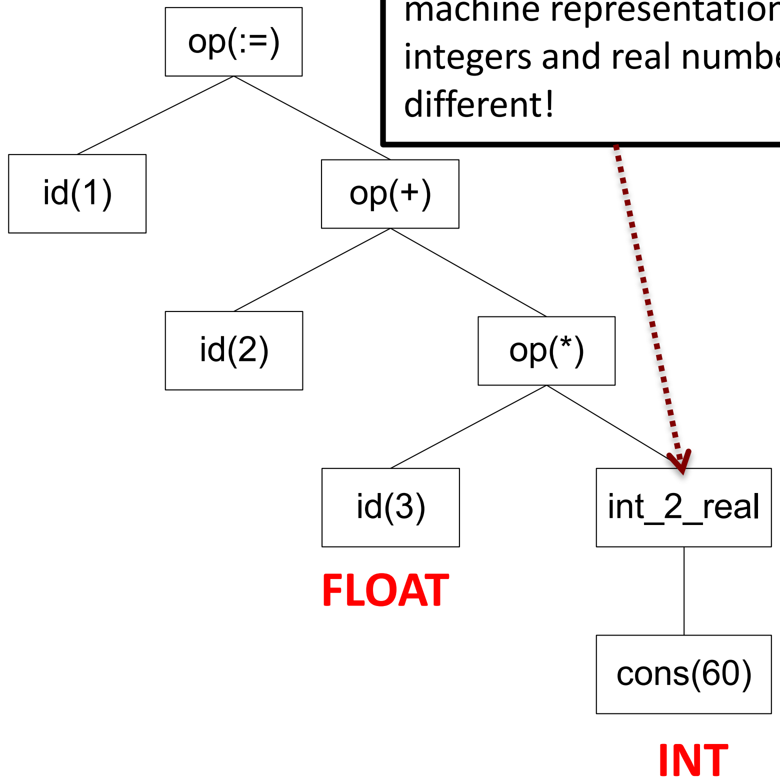
$expression_1 * expression_2$

$(expression_1)$

Semantic Analysis

- Checks the program for semantic errors
- Gathers type information
- Operands and operators
- Type-checking

int_2_real() is an extra node for converting **60** to a real number. **Remember**: the machine representation of integers and real numbers is different!




```
position := initial + rate * 60
```

Error detection and reporting

- All phases can issue errors
- A compiler that stops at the first error is not helpful
- Most of the errors are handled in the syntax/semantic analysis phases
  - Lexical analysis detects errors where a stream of characters does not form a valid token
  - Syntax analysis detects errors where the stream of valid tokens violate the structure rules (syntax)
  - Semantic analysis detects errors where the syntax is valid by the operation not (adding an array with a real number)

Intermediate code and optimization

- Each phase produces intermediate code

```
temp1 := int_2_real(60)
temp2 := id(3) * temp1
temp3 := id(2) + temp2
id(1) := temp3
```

three-address code: a simple assembly-like language, which consists of instructions, each of which has at most three operands

- Optimization

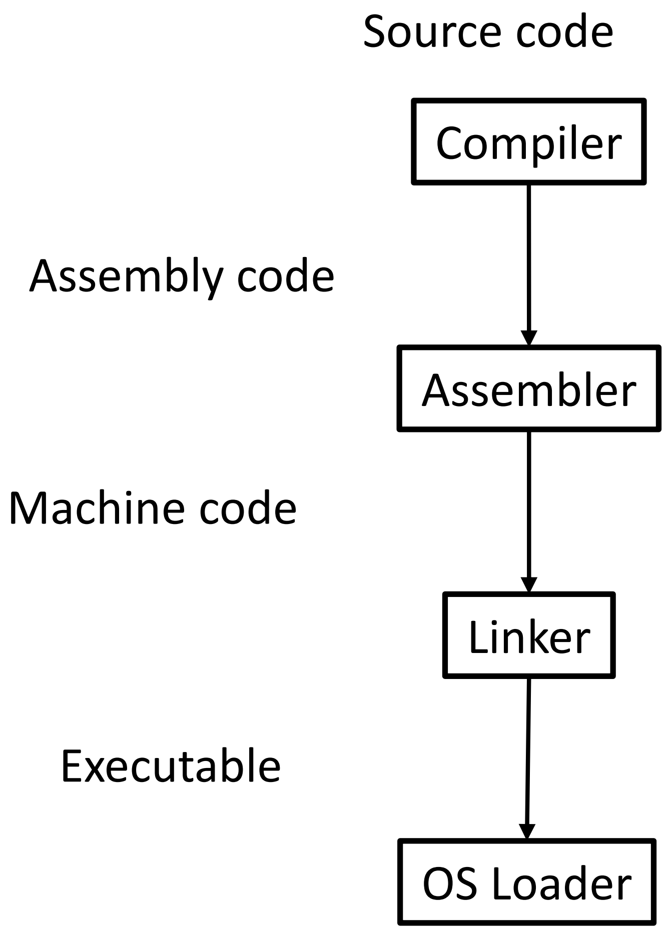
```
temp1 := id(3) * 60.0
id(1) := id(2) + temp1
```

Code generation

- The last phase of the compiler is the generation of the target code
- Register allocation
  – Each expression should use registers that are available
- Relocation information
  – Variables are stored in relocatable addresses

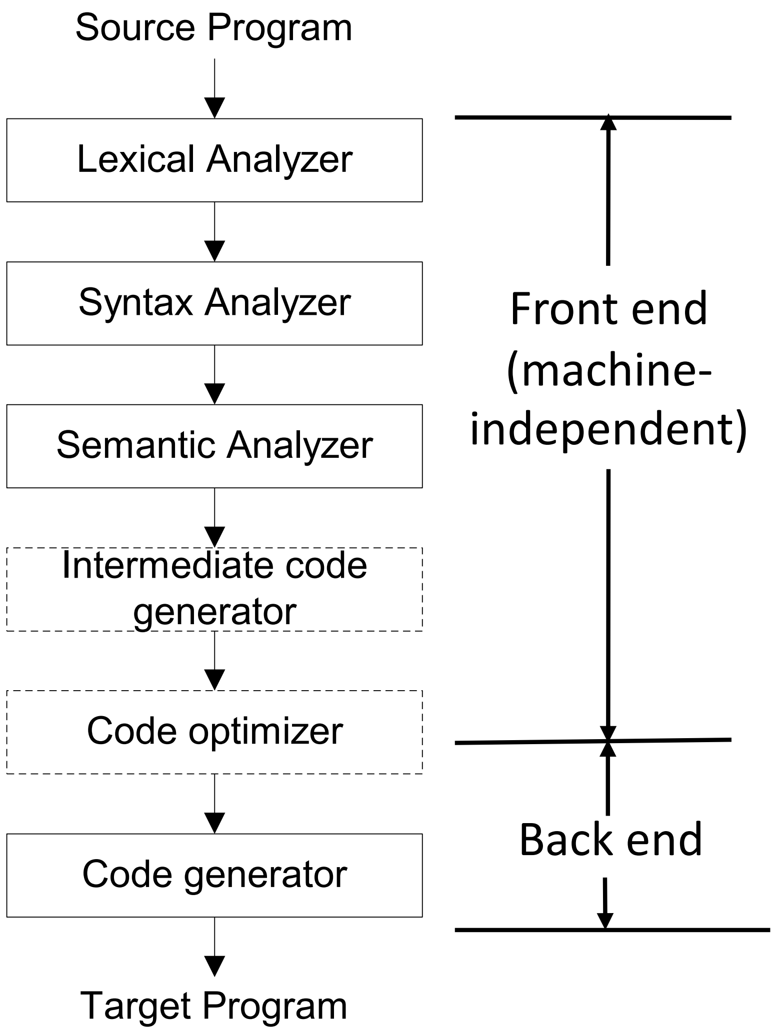
```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Compiler pipeline



Front and back ends

- Separation of common tasks
- Makes design and implementation easier
- K compilers for N machines
  - N back ends, K front ends
  - Instead of K*N compilers



Passes

- A *pass* is when the compiler reads the source code (or intermediate files)
- The number of passes depends on the source and target language and the running environment
- Different phases that cooperate can be grouped to a single pass (not always possible)
- When grouping is not possible
  – **Backpatching**: leave empty information that is going to be filled by a later phase/pass

Compiler-construction Tools

- Parser generators
- Scanner generators
- Syntax-directed translation engines
- Automatic code generators
- Data-flow engines