Introduction

Programming a Computer

- High level language (HLL) program: easy for human understanding
- Assembly language program: intermediate form.
- Machine language program: a CPU can only understand this form.

A Compiler

- A compiler is a program that accepts a HLL program (source language) as input. It translates the program to a target language program preserving the semantics.
- An example the source language may be C++ and the target language may be the assembly or machine language of some processors e.g. x86-64.

Compiling a Compiler

- A compiler is also a program written in some language and compiled. The implementation language of a compiler may or may not be same as its source language.
- If the source language and the implementation language are the same, we are essentially compiling a new version of the compiler. A process known as bootstrapping.

An Interpreter

- An interpreter is a type of compiler that works in a different mode.
- It does not produce a complete translated version of the machine code. It interprets the extracted semantic representation of the HLL program, and performs necessary action on data.

C Program

```
#include <stdio.h>
int main() // first0.c
{
    printf("My first program\n");
    return 0;
}
```

Assembly Language Program

`$ cc -Wall -S first0.c` ⇒ `first0.s`

```
    .file   "first0.c"
    .section    .rodata
.LC0:
    .string   "My first program"
    .text
    .globl   main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
```

```
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq   %rsp, %rbp
    .cfi_def_cfa_register 6
    movl   $.LC0, %edi
    call   puts
    movl   $0, %eax
    popq   %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident   "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
```

```
.section   .note.GNU-stack,"",@progbits
```

Object File

$ cc -c first0.s ⇒ first0.o

$ objdump -d first0.o | less

```
0000000000000000 <main>:
   0: 55                  push   %rbp
   1: 48 89 e5            mov    %rsp,%rbp
   4: bf 00 00 00 00      mov    $0x0,%edi
   9: e8 00 00 00 00      callq  e <main+0xe>
   e: b8 00 00 00 00      mov    $0x0,%eax
  13: 5d                  pop    %rbp
  14: c3                  retq
```

Executable File

`$ cc first0.o` ⇒ `a.out`

```
$ objdump -d a.out | less
```

```
00000000004004f4 <main>:
  4004f4:   55                  push   %rbp
  4004f5:   48 89 e5            mov    %rsp,%rbp
  4004f8:   bf fc 05 40 00      mov    $0x4005fc,%edi
  4004fd:   e8 ee fe ff ff      callq  4003f0 <puts@plt>
  400502:   b8 00 00 00 00      mov    $0x0,%eax
  400507:   5d                  pop    %rbp
  400508:   c3                  retq
```

`a.out` file contains more code than this.

Using Software Interrupt: x86-64

```
#include <asm/unistd.h>
#include <syscall.h>
#define STDOUT_FILENO 1

.file "first.S"
.section        .rodata
L1:
   .string "My First program\n"
L2:
.text
.globl _start
```

```
_start:
  movl  $(SYS_write), %eax      # eax <-- 1 (write)
                                # parameters to 'write'
  movq  $(STDOUT_FILENO), %rdi  # rdi <-- 1 (stdout)
  movq  $L1, %rsi               # rsi <-- starting
                                #  address of string
  movq  $(L2-L1), %rdx          # rdx <-- L2 - L1
                                #   string length
  syscall                       # software interrupt
                                #  user process requesting
                                #  OS for service
  movl  $(SYS_exit), %eax       # eax <-- 60 (exit)
                                #   parameters to exit
```

```
movq  $0, %rdi                    # rdi <-- 0
syscall                           # software interrupt

ret                               # return
```

Preprocessor, assembler and Linker

```
$ /lib/cpp first.S first.s
$ as -o first.o first.s
$ ld first.o
$ ./a.out
My first program
```

Description/Specification of a Language

- Description of a well-formed program - syntax of a language.
- Description of the meaning of different constructs and their composition as a whole program - semantics of a language.

Description of Syntax

Syntax of a programming language is specified and verified in two stages.

1. Identification of the tokens (atoms of different syntactic categories) from the character stream of a program.

2. Correctness of the syntactic structure of the program from the stream of tokens.

Description of Syntax

Formal language specifications e.g. regular expression, formal grammar, automaton etc. are used to specify the syntax of a language.

Description of Syntax

Regular language specification is used to specify different syntactic categories.

Restricted subclass of the context-free grammar e.g. LL(1), LALR(1), or LR(1) are used to specify the structure of a syntactically correct program.

Note

There are structural features of a programming language that are not specified by the grammar rules for efficiency reason and are handled differently.

Description of Meaning: Semantics

- Informal or semi-formal description by natural language and mathematical notations.
- Formal descriptions e.g. grammar rule with attributes, different formal specifications of semantics.

Users of Specification

- Programmer - often uses an informal description of the language construct.
- Implementer of the language translator for a target machine or language.
- People who want to verify a piece of program or who want to automate program writing (synthesis).

Source and Target

A source language is usually a high-level language. But there are different types of high level languages.
Imperative languages like C, object oriented languages like Java, functional languages like Haskell, logic programming languages like Prolog, languages for parallel and distributed programming etc.

Source and Target

- The target languages also have a wide spectrum. Assembly and machine languages of different architecture, some high-level language e.g. C, languages of virtual machines etc.
- Variation in machine architecture puts different code improvement demand on a compiler.

Front-end and Back-end

- The part of the compiler that analyzes the structure of the source program, extracts the semantic information and produces an internal representation of it is known as the front-end.
- The part that uses the semantic representation and synthesis the semantically equivalent target program is called the back-end.

Compiler and Interpreter

- A compiler and an interpreter mainly differ in their back-ends.
- A compiler from the intermediate representation try to generate a target code that will run efficiently, many times, on different data.

Compiler and Interpreter

- Interpreter on the other hand will perform action specified by the HLL program fragment, extracted in the intermediate representation, on its data.
- The code generation back-end of an compiler is replaced by a set of routines for interpretation.

Compiler and Interpreter

- Usually it is expected that the compiled code is more time efficient.
- But an interpreter may have better error reporting as the source program is available.
- It may be more portable and easier to write. People also claim that an interpreter is better in terms of security.

Basic Phases of Compilation

- Read the program text - this is the most time consuming part as it involves I/O.
- Preprocessing - a phase before the actual compilation. It may involve inclusion (reading) of several files.
- Lexical analysis - identification of the syntactic symbols of the language and collecting their attributes.

Basic Phases of Compilation

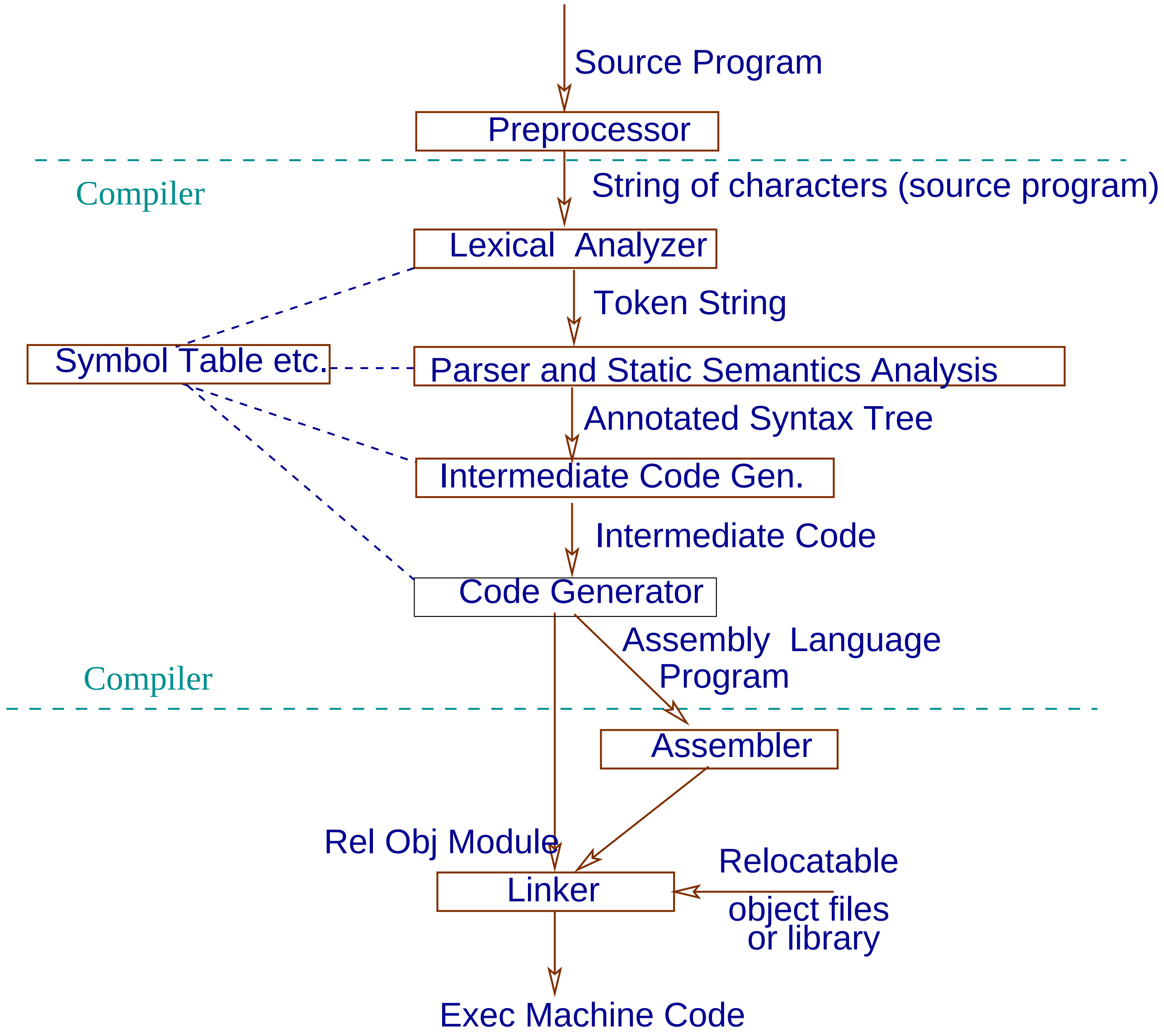
- Syntax checking and static semantic analysis - the stream of token is parsed to form the parse tree or syntax tree. The semantic information is collected from the context and the syntax tree is annotated.

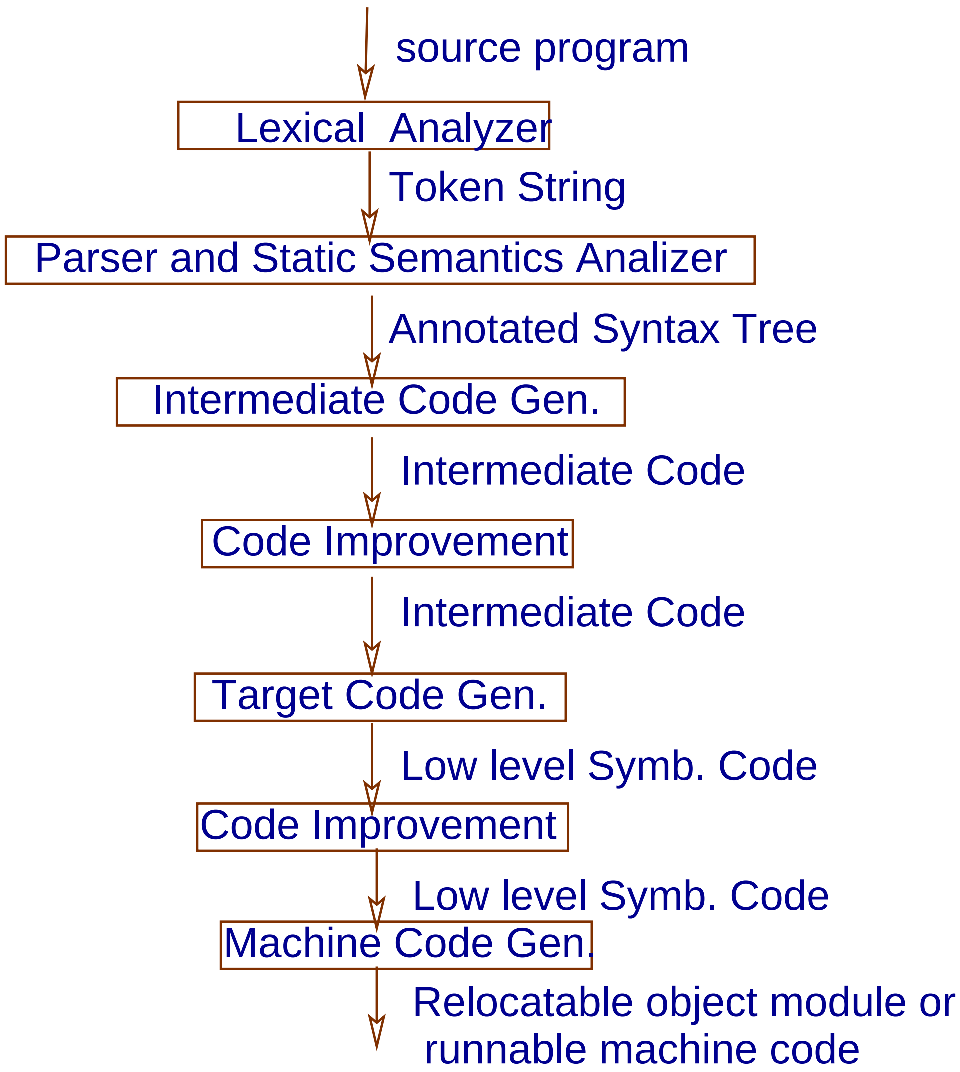
Basic Phases of Compilation

- Intermediate code generation and code improvement - language specific constructs are translated to more general and simple constructs. As an example a long expressions is broken down to expressions with fixed number of parameters. Different optimizations are performed on this intermediate code.

Basic Phases of Compilation

- Symbolic target code generation and architecture specific code improvement - the intermediate representation is translated to target code in symbolic form, and architecture specific code optimization is performed.
- Low level machine code generation.


Source Program
Preprocessor
Compiler
String of characters (source program)
Lexical Analyzer
Token String
Symbol Table etc.
Parser and Static Semantics Analysis
Annotated Syntax Tree
Intermediate Code Gen.
Intermediate Code
Code Generator
Assembly Language Program
Compiler
Assembler
Rel Obj Module
Relocatable object files or library
Linker
Exec Machine Code


source program
Lexical Analyzer
Token String
Parser and Static Semantics Analizer
Annotated Syntax Tree
Intermediate Code Gen.
Intermediate Code
Code Improvement
Intermediate Code
Target Code Gen.
Low level Symb. Code
Code Improvement
Low level Symb. Code
Machine Code Gen.
Relocatable object module or
runnable machine code

Independence of Front and back Ends

- In an idealistic situation the front-end of a compiler does not know anything of the target language. Its job is to transform the source program to intermediate representation.
- Similarly, the back-end is not aware of the source language. It works on the intermediate representation to generate the target code.

Compiler and Interpreter

- If the intermediate representation of the source program and the input data to it is available, the ‘compiler’ can perform the action on the data specified by the source program using the intermediate representation. There is no need to generate the target code
- This is what is done by an interpreter.

Scanner or Lexical Analyzer

A scanner or lexical analyzer breaks the program text (string of ASCII characters) into the alphabet of the language (into syntactic categories) called a tokens.

A token is a symbol of the alphabet of the language. It may be encoded as a number. A token may have one or more attributes.

An Example

Consider the following C function.

```
double CtoF(double cel) {
        return cel * 9 / 5.0 + 32 ;
}
```

Scanner or Lexical Analyzer

The scanner uses the finite automaton model to identify different tokens. Software are available that takes the specification of the tokens (elements of different syntactic categories) in the form of regular expressions and generates a program that works as the scanner. The process is completely automated.

Scanner or Lexical Analyzer

- A syntax analyzer does not differentiate between different identifiers or different integer constants. So they are identified as identifier token and integer token.
- But the actual values are preserved as attributes for use in subsequent phases.

Syntactic Category, Token and Attribute

String	Type	Token	Attribute
"double"	keyword	302	
"CtoF"	identifier	401	"CtoF"
"("	delimiter	40	
"double"	keyword	302	
"cel"	identifier	401	"cel"
")"	delimiter	41	
"{"	delimiter	123	
"return"	keyword	315	

String	Type	Token	Attribute
“cel”	identifier	401	“cel”
“*”	operator	42	
“9”	int-numeral	504	9
“/”	operator	47	
“5.0”	double-numeral	507	5.0
“+”	operator	43	
“32”	int-numeral	504	32
“;”	delimiter	59	
“}”	delimiter	125	

Parser or Syntax Analyzer

A parser or syntax analyzer checks whether the token string generated by the scanner, forms a valid program. It uses a restricted class of context-free grammars to specify the language constructs.

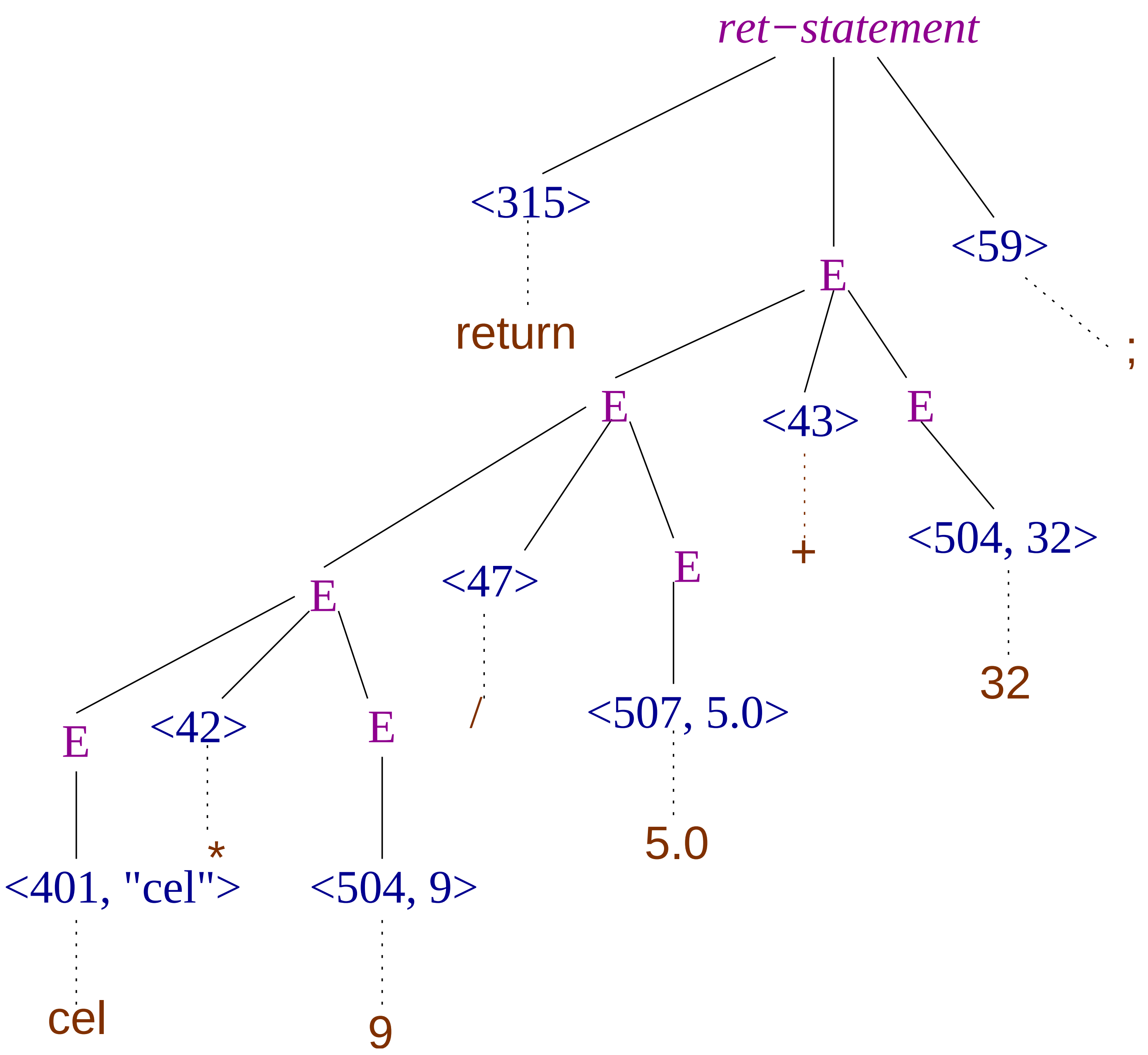
Context-Free Grammar

$$
\begin{array}{rcl}
\text{function-definition} & \rightarrow & \text{decl-spec decl comp-stat} \\
\text{decl-spec} & \rightarrow & \text{type-spec} \mid \cdots \\
\text{type-spec} & \rightarrow & \texttt{double} \mid \cdots \\
\text{decl} & \rightarrow & \text{d-decl} \mid \cdots \\
\text{d-decl} & \rightarrow & \text{ident} \mid \text{ident } \texttt{(} \text{ par-list } \texttt{)} \\
\text{par-list} & \rightarrow & \text{par-dcl} \mid \cdots \\
\text{par-dcl} & \rightarrow & \text{decl-spec decl} \mid \cdots
\end{array}
$$

Expression Grammar

$$E \;\rightarrow\; E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid$$
$$-E \mid \text{var} \mid \text{float-cons} \mid \text{int-cons} \mid \cdots$$

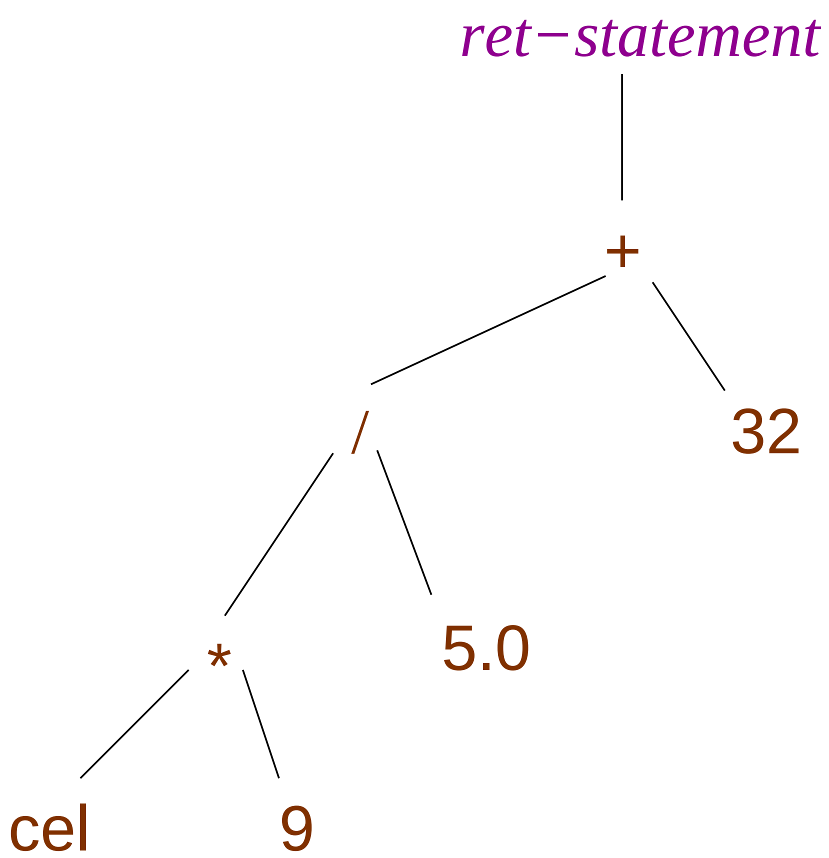
Parse Tree


ret−statement
<315>
return
E
<59>
;
E
<43>
E
+
<504, 32>
32
E
<47>
E
/
<507, 5.0>
5.0
E
<42>
E
*
<401, "cel">
cel
<504, 9>
9

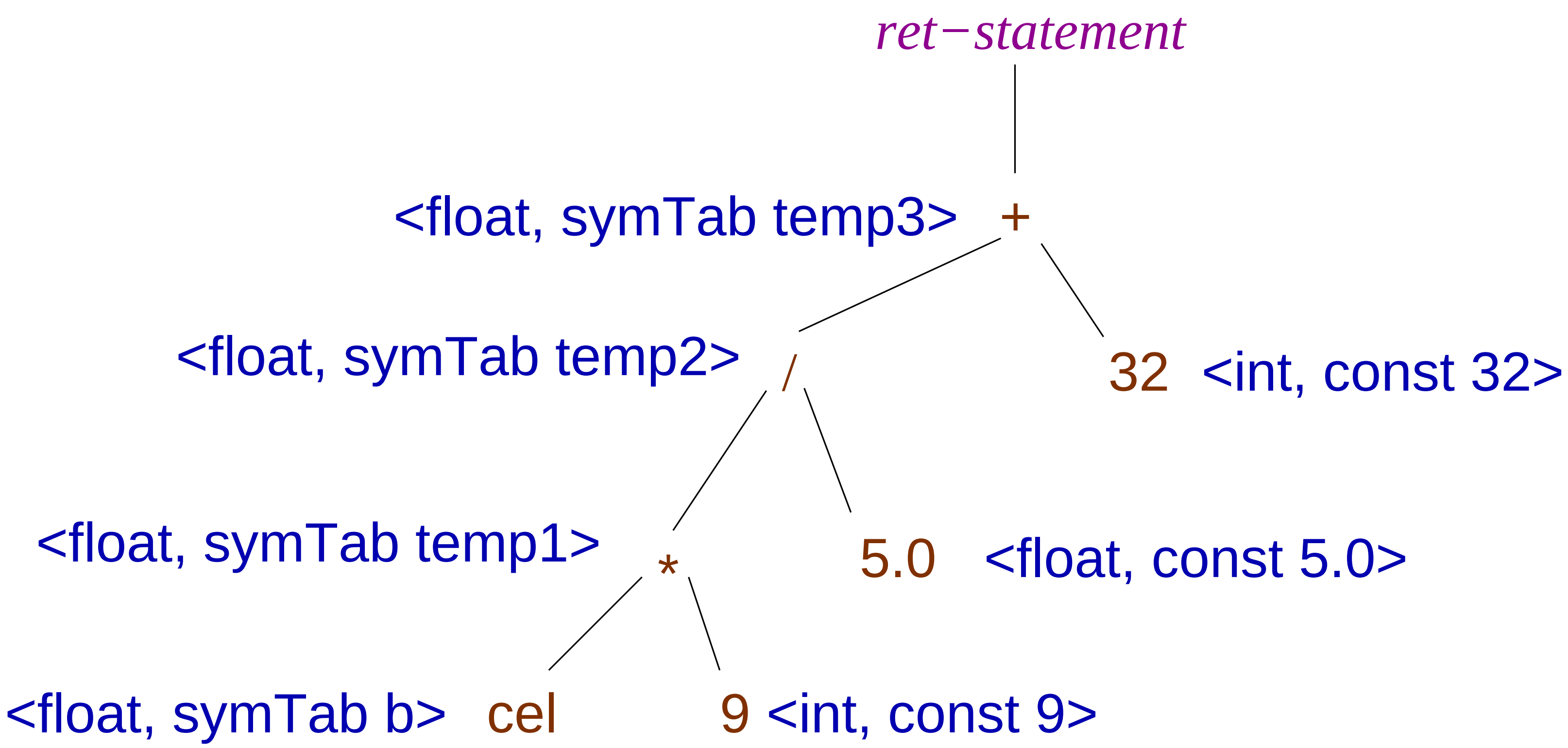
Abstract Syntax Tree

- The parse tree generated during syntax checking is not suitable a representation for further processing. A modified form, known as abstract syntax tree (AST) is created.
- Semantic information are stored in the nodes of AST (annotated AST) as attributes. It is used for intermediate code generation, error checking and code improvement.

Abstract Syntax Tree (AST)



Annotated AST



Symbol Table

The compiler maintains an important data structure called the symbol table to store variety of names and their attributes it encounters. A few examples are - variables, named constants, function names, type names, labels etc.

Semantic Analysis

The symbol table corresponding to the function `CtoF` should have an entry for the variable `cel` with its type and other information.
The constant 9 is of type `int`. It is to be converted to 9.0 of type `double` before it can be multiplied with `cel`. Similar is the case for 32.

Semantic Analysis

It is not enough to know that `x = x + 5;` is syntactically correct.

- The operation is meaningful only if the variable `x` is declared, it is a number, and not a string etc.
- Even if `x` is a number, the generated code will be different depending on whether it is an `int`, `float` or a `pointer`.

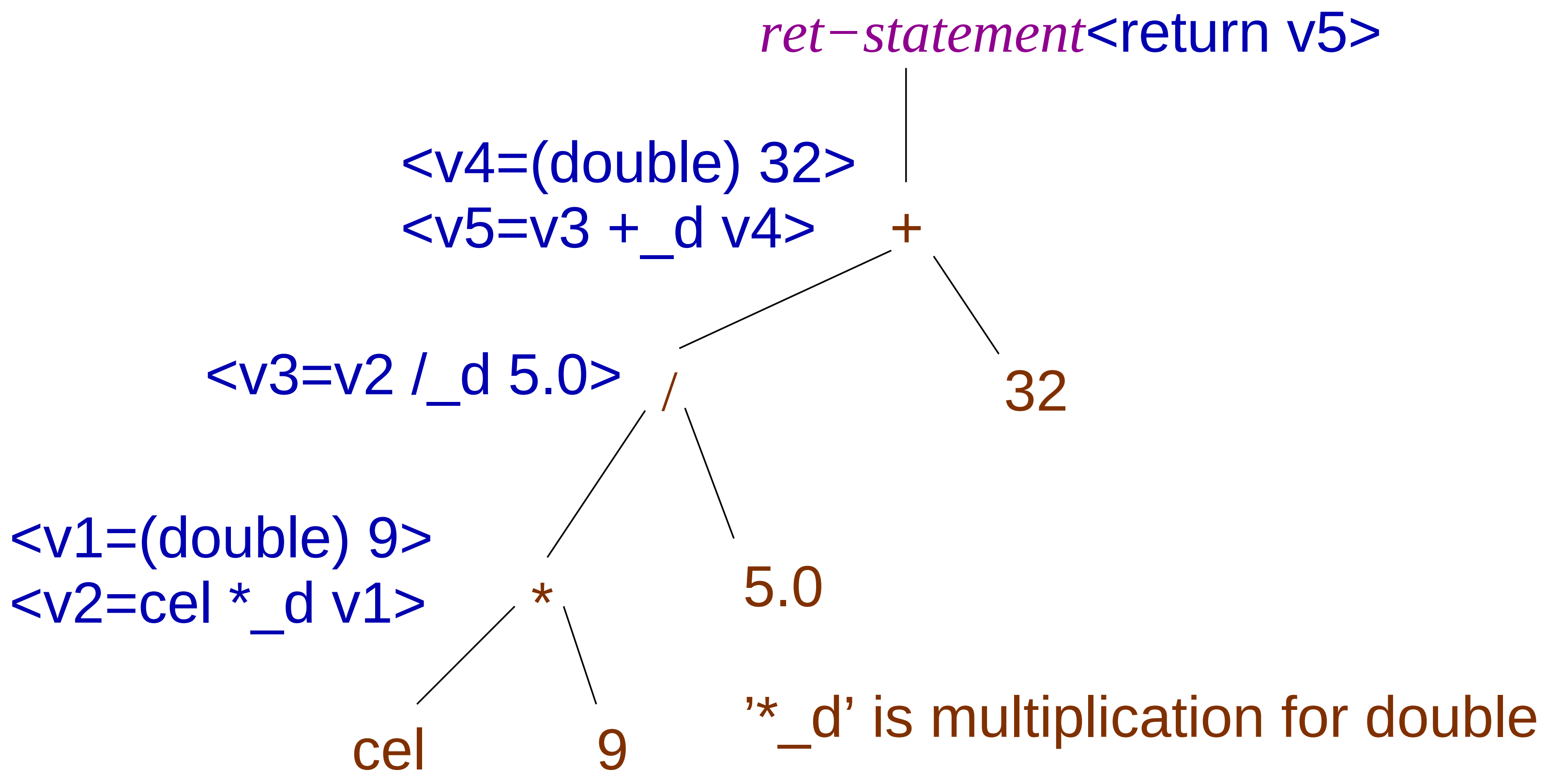
Intermediate Code Generation

- The language specific AST is translated into more general constructs known as intermediate code e.g. 3-address code.
- The form of the intermediate code should be suitable for code improvement and target code generation.

Intermediate Code Generation

- A while-loop in a C-like language may be replaced by test, and conditional and unconditional jumps.
- A compiler may use more than one intermediate representations for different phases.

Intermediate Code



Intermediate Code

param cel

v1 = (double) 9 # compile time

$v2 = cel *_d v1$

$v3 = v2/_d 5.0$

v4 = (double)32 # compile time

$v5 = v3 +_d v4$

return v5

Note

`v1, v2, v3, v4, v5` are called virtual registers. Finally they will be mapped to actual registers or memory locations. The distinct names of the virtual registers help the compiler to improve the code.

GCC IC - GIMPLE

```
$ cc -Wall -fdump-tree-gimple -S ctof.c
```

```
CtoF (double cel)
{
  double D.1796;

  _1 = cel * 9.0e+0;
  _2 = _1 / 5.0e+0;
  D.1796 = _2 + 3.2e+1;
  return D.1796;
}
```

Raw GIMPLE Code

```
$ cc -Wall -fdump-tree-gimple-raw -S ctof.c

CtoF (double cel)
gimple_bind <
  double D.1796;

  gimple_assign <mult_expr, _1, cel, 9.0e+0, NULL>
  gimple_assign <rdiv_expr, _2, _1, 5.0e+0, NULL>
  gimple_assign <plus_expr, D.1796, _2, 3.2e+1, NULL>
  gimple_return <D.1796 NULL>
>
```

Intermediate Code Improvement

- Different code improvement transformations are performed on the intermediate code.
- A few examples are constant propagation, constant folding, strength reduction, copy propagation, elimination of common sub-expression, $\cdots$, code in-lining etc.

Target Code Generation

- Program variables and temporary variables are allocated to memory and registers.
- Translates the intermediate code to a target language code e.g. sequence of assembly language instructions of a machine.

Target Code Improvement

- The sequence of target code e.g. assembly language code of an architecture may be modified to improve the quality of code.
- It may replace a sequence of instructions by a better sequence to make the code faster on an architecture.

64-bit Intel Code

```
    .file   "ctof.c"
    .text
.globl CtoF
    .type   CtoF, @function
CtoF:
.LFB2:
    pushq   %rbp                # save old base pointer
.LCFI0:
    movq    %rsp, %rbp          # rbp <-- rsp new base pointe
.LCFI1:
    movsd   %xmm0, -8(%rbp)     # cel <-- xmm0 (parameter)
    movsd   -8(%rbp), %xmm1     # xmm1 <-- cel
```

```
    movsd   .LC0(%rip), %xmm0 # xmm0 <--- 9.0, PC relative
                              # addressing of read-only dat
    mulsd   %xmm0, %xmm1      # xmm1 <-- cel*9.0
    movsd   .LC1(%rip), %xmm0 # xmm0 <-- 5.0
    divsd   %xmm0, %xmm1      # xmm1 <-- cel*9.0/5.0
    movsd   .LC2(%rip), %xmm0 # xmm0 <-- 32.0
    addsd   %xmm1, %xmm0      # xmm0 <-- cel*9.0/5/0+32.0
                              # return value in xmm0
    leave
    ret
.LFE2:
    .size   CtoF, .-CtoF
    .section        .rodata
    .align 8
```

```
.LC0:
    .long   0
    .long   1075970048
    .align 8
.LC1:
    .long   0
    .long   1075052544
    .align 8
.LC2:
    .long   0
    .long   1077936128
```

9.0 in IEEE-754 Double Prec.

63

0 | 1000 0000 010 | 0010 0000 0000 0000 0000

31

0000 0000 0000 0000 0000 0000 0000 0000

Exponent Bias: 1023,
Actual exponent: $1026 - 1023 = 3$.
$9.0_{10} = 1001.0_2 = 1.001 \times 2^3$.

9.0 and `.LC0`

Interpreted as integer we have the higher order 32-bits as $2^{30} + 2^{21} + 2^{17} = 1075970048$ and the lower order 32-bits as 0.

In the little-endian (lsb) data storage, lower bytes comes first.

9.0 and .LC0

```
    .align 8
.LC0:
    .long   0
    .long   1075970048
```

is 9.0.

Improved Code `$ cc -Wall -S -O2 ctof.c`

```
    .file   "ctof.c"
    .text
    .p2align 4,,15
.globl CtoF
    .type   CtoF, @function
CtoF:
.LFB2:
    mulsd   .LC0(%rip), %xmm0
    divsd   .LC1(%rip), %xmm0
    addsd   .LC2(%rip), %xmm0
    ret
.LFE2:
```

```
    .size   CtoF, .-CtoF
    .section        .rodata.cst8,"aM",@progbits,8
    .align 8
.LC0:
    .long   0
    .long   1075970048
    .align 8
.LC1:
    .long   0
    .long   1075052544
    .align 8
.LC2:
    .long   0
    .long   1077936128
```