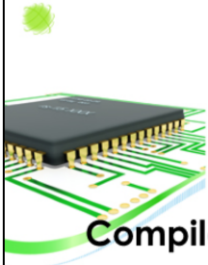**Language Processors**

Introduction

Definition

A language processor is a tool
that translates formal inputs,
which conform to a language, to
target representations intended.

The formal input conforms a predefined set of rules that was tailored to serve the solution of a problem that belongs to a particular domain. Generally, the domain is software engineering, the problem is programming, the rule set is a programming language, the formal input is a program, and the target representation is an executable code.

An example for this general understanding can be given by using C++ as a central concept. The domain is software engineering; the problem is imperative programming; the programming language is C++; any C++ source code is formal input. Depending on the targets designated, the output can be a file that belongs to wide range of types including but not limited to .exe, .dll, .dylib, .lib .

Another partially non-conforming example can be given with reference to another so called industrial standard: XML. In this case, the domain is software engineering; the problem is document (or speaking generally data) representation, the rule set is XML language specification; the language processor is XML processor. Any XML document can be an input. The output is application specific and virtually infinite. An example can be transient memory representation of a parsed SOAP message during processing of a business transaction.

Widely Referred Types

Compiler

Simply stated, a compiler is a program that can read a program in one language – the source language – and translate it into an equivalent program in another language – the target language.

Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book

A compiler is simply a computer program that translates other computer programs to prepare them for execution.

Cooper, K.D., Torczon, L.; Engineering A Compiler

Interpreter

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book

An Interpreter takes as input an executable specification and produces as output the result of executing the specification.

Cooper, K.D., Torczon, L.; Engineering A Compiler

Both the compilers and the interpreters have the ability of understanding their inputs. The basic difference lays in the phase of the execution. The interpreter executes the source code that was understood. The compiler on the other hand generates the translated code leaving the execution to another layer of the architecture of computation.

There are many sources that make mention of assemblers and translators, which are essentially compilers that generate intended target representations. The resources counting either or both of these next to the compilers regard the discrimination of language levels. The assemblers process programs written in assembly language and generate machine code. Assembly is the low level language which is at the closest level to the machine instruction codes. So, the translation is from low level to low level. The case for the translators is from high level source to another high level target even though the target can be at some lower level in relation to the level of the source.
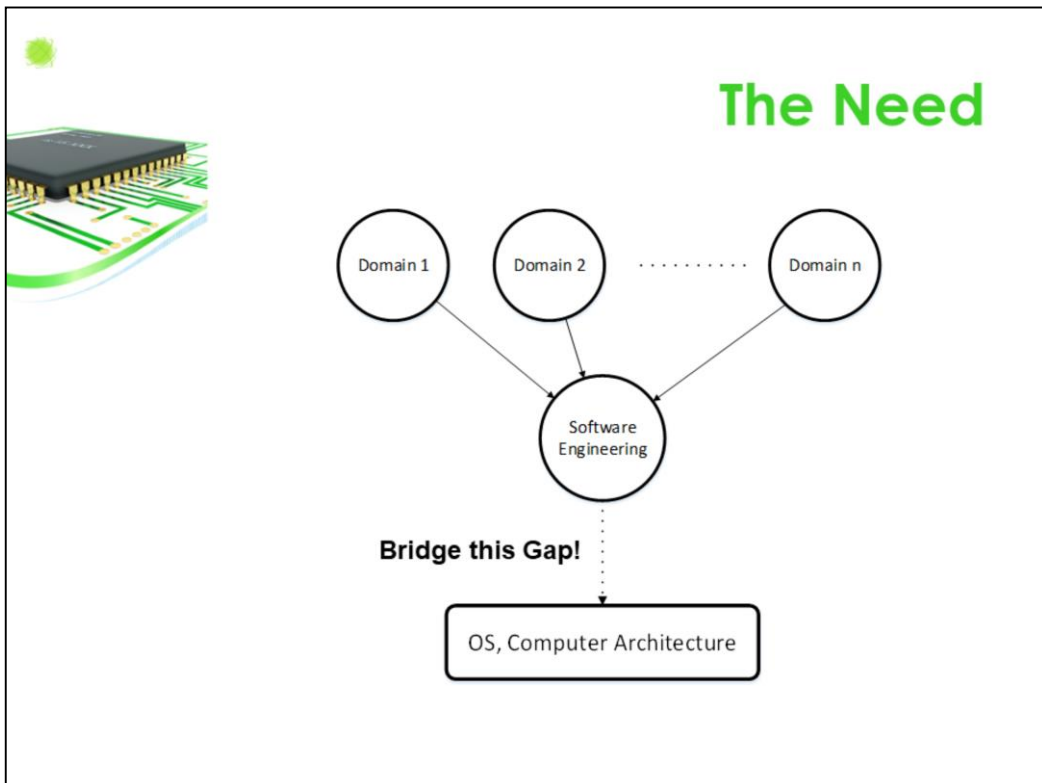
The fast answer is that we need to develop complete architecture for computations to serve needs of various applications.

The language processors are useful tools that help us bridge the gap between the computation models that address needs of application domains at one end and the underlying system architectures at the other. Bridging the gap requires sound understanding of both the needs and the architecture of the respective ends.

It is generally correct to state that the specification of computations in terms of system level constructs are more complex then those made in terms of higher level constructs such as commonly known programming languages. The complexity manifests itself in various forms including but not limited to composition, maintenance, portability, and reusability. A concrete example is the arithmetic expression pattern which is known by CS students since the beginning of their education. An arithmetic expression is a higher level construct that is mapped into a series of processor instructions by the compiler that was developed with the underlying instruction set architecture in mind.

Software engineering, being at a very critical point for provisioning of computing power to serve infinitely many needs arising from various domains,

has to keep up with endless demands under many pressures like time, quality, performance, customer satisfaction, and more. The language processors come in handy because they provide the engineering team with an abstraction having following benefits.

i.   Thinking in terms of higher level constructs.

ii.   Trouble free use of underlying architecture by sound translation to lower level constructs.

iii.   Higher possibility of cross platform compatibility.

iv.   Reusability with lower effort.

v.   Safety to the extent provided by the language design.

Any attempt to create a timeline of the programming languages emerged so far will detect at least hundreds of projects. Even when focused to the imperative languages, which belong one subset of language processors, the list will contain at least 400 entries according to the Wikipedia page created for the timeline of the programming languages.

Why do we create a new programming language? It is possible to find evolutionary, technical, practical, commercial, and even politic-economical reasons for this. With the advent of the object orientation as a design and programming concept, object oriented languages emerged as an improvement over existing programming languages. This can be regarded as an evolutionary reason. C++ emerged to address the concern of execution speed while adhering to object oriented concepts. On the other hand, to reduce the burden of resource management and keeping it object oriented, Java has become a good choice. In this regard, we can safely state that C++ and Java has come to existence for good technical reasons. We know that there are practical reasons for using PHP (Personal Hypertext Preprocessor). It mixes the HTML (Hypertext Markup Language) with imperative statements with a style that creates flexibility enough for generation of dynamic web pages. It is interpreted safely, it does not require heavy "out of process" execution of toolchains to get the work done. Let's focus on Microsoft's Visual Basic for instance. This example combines both practical and commercial reasons. Visual Basic resided in commercial confines for many years mainly

to support the company's office productivity tools such as MS-Excel, MS-Word. The macro language required to automate user-tasks expressed, stored, circulated with the name of this commercially branded language, which helped Microsoft keep its competitive edge for a very long period. Commercialization is the point where the job of language design and development of related language processors starts to get ugly. This is at the fringes of a domain where economical concerns are assumed as superior to technical concerns.

Yet Another One?

New languages, new language processors…
Do we really need to invent a new one?

Reasons

- … and even political-economic

  "Much of the past Soviet systems software effort has been in programming languages. This is reflected in the large proportion of the open publications devoted to this area, and is consistent with the given hardware constraints, the relatively formal academic orientation of Soviet software research personnel, and the historical pattern followed in the West. Something like 50 different higher level languages can be identified from the literature. Many are experimental and have had virtually no impact beyond their development groups."

  Goodman, S. E. (1979). Software in the Soviet Union: Progress and Problems. Advances in Computers Volume 18, 231–287. doi:10.1016/s0065-2458(08)60585-9

It will not take long the political minds to discover that the knowledge of developing languages, like any technical knowledge, is related to power and control. In order to go beyond speculation and to find concrete examples, we can see the literature on development of Soviet history of computing.

It is known that as of Russians used a localized version of Algol, ALGEM, to fit the limitations of the hardware and to address the lingua-cultural concerns on identifiers coded in Russian.

As a last remark, we need to keep in mind that when it comes to deciding to create a new language processors, most probably the reasons will come in combination not in isolation.

What's in the Box?

Compiler as an example ...
Details according to the need and taste.
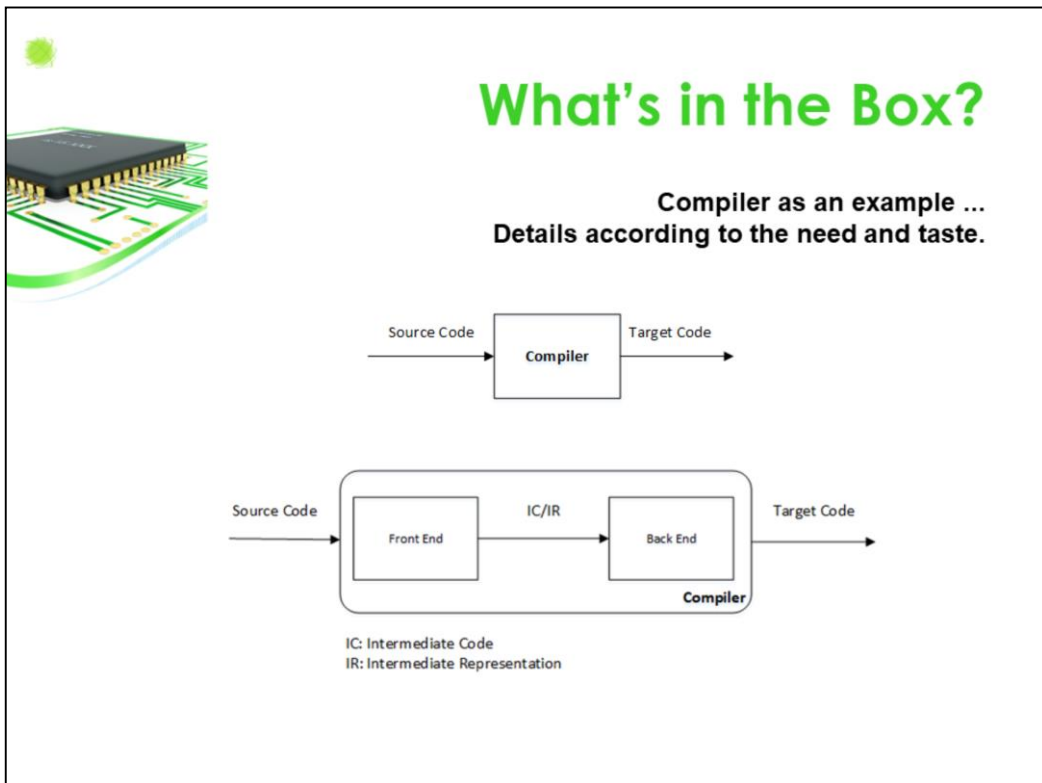
A language processor must have the capacity to check conformance of its input to the formal and the semantic requirements as specified by the language definition. Further to this, it must have the capacity to perform all necessary translations to synthesize target representations and / or to perform all necessary computations with strict adherence to the language specification. A good language processor must be sensitive to the probable errors in its input, recover from them when possible, and report them to the supplier of the input.

Let's see a few "box models" that are valid for the compilers, which are the language processors we will focus in this course. In the first model, the compiler is a single magic box that receives the source code and generates the target code. Following models are developed by the authors to convey a bit more detail.

Diagrams are from "Cooper, K.D., Torczon, L.; Engineering A Compiler" and "Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book"

What's in the Box?
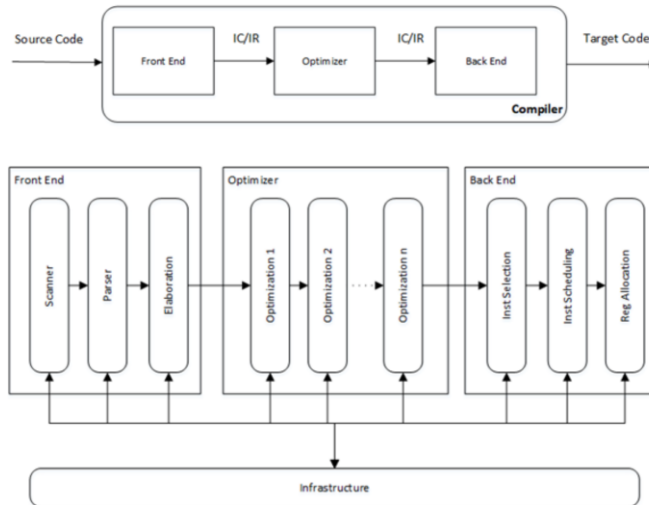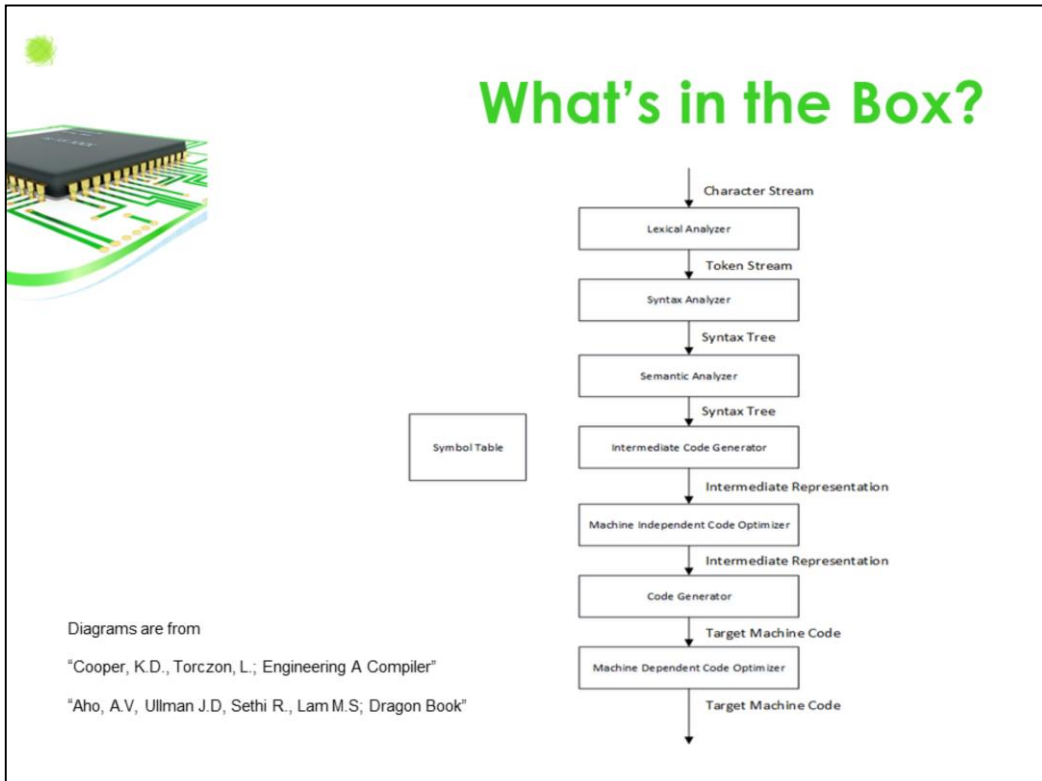
Any of the diagrams shown so far are valid. More valid diagrams can be built and presented. The key take away is that any of these diagrams are "compatible" to the definition of a language processor! The stages in the diagrams are all valid but some of them can be omitted by implementations.
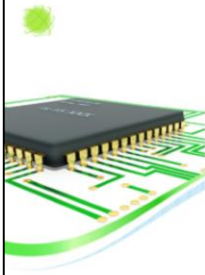
Architecture

- Programmers see the computer through the window provided by the designers at the level of its functional architecture.
- This window is provided by you, the designer. (1)

  ❧The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation. (2)

1. From Prof. Dr. Bozşahin's CENG444 course lecture.
2. Amdahl, Gene & Blaauw, Gerrit & Brooks, Jr, Frederick. (2000). Architecture of the IBM System/360. IBM Journal of Research and Development. 44. 21-36. 10.1147/rd.82.0087.

Architecture

int f(int p)
{
    if (p>1)
        return p*f(p-1);

    return 1;
}

Identical semantics mapped different
Instruction Set Architectures (ISA)

Translation requires sound knowledge on
the target ISA

```
00007FF79EF52320 89 4C 24 08      mov     dword ptr [rsp+8],ecx
00007FF79EF52324 55               push    rbp
00007FF79EF52325 57               push    rdi
00007FF79EF52326 48 81 EC E8 00 00 00 sub  rsp,0E8h
00007FF79EF5232D 48 8D 6C 24 20    lea     rbp,[rsp+20h]
00007FF79EF52332 48 8D 0D F0 0C 01 00 lea  rcx,[__F791F1DE_Lecture01@cpp (07FF79EF63029h)]
00007FF79EF52339 E8 AA F0 FF FF    call    __CheckForDebuggerJustMyCode (07FF79EF513E8h)
    if (p>1)
00007FF79EF5233E 83 BD E0 00 00 00 01 cmp  dword ptr [p],1
00007FF79EF52345 7E 1C            jle     f+43h (07FF79EF52363h)
    return p*f(p-1);
00007FF79EF52347 8B 85 E0 00 00 00 mov     eax,dword ptr [p]
00007FF79EF5234D FF C8            dec     eax
00007FF79EF5234F 8B C8            mov     ecx,eax
00007FF79EF52351 E8 75 EF FF FF   call    f (07FF79EF512CBh)
00007FF79EF52356 8B 8D E0 00 00 00 mov     ecx,dword ptr [p]
00007FF79EF5235C 0F AF C8         imul    ecx,eax
00007FF79EF5235F 8B C1           mov     eax,ecx
00007FF79EF52361 EB 05           jmp     f+48h (07FF79EF52368h)
}
    return 1;
00007FF79EF52363 B8 01 00 00 00   mov     eax,1
}
00007FF79EF52368 48 8D A5 C8 00 00 00 lea  rsp,[rbp+0C8h]
00007FF79EF5236F 5F              pop     rdi
00007FF79EF52370 5D              pop     rbp
00007FF79EF52371 C3              ret
```

x64 - CISC

```
0x10000309c (+0):  sub    sp, sp, #0x20
0x10000030a0 (+4):  stp    x29, x30, [sp, #0x10]
0x10000030a4 (+8):  add    x29, sp, #0x10
0x10000030a8 (+12): str    w0, [sp, #0x8]
0x10000030ac (+16): ldr    w8, [sp, #0x8]
0x10000030b0 (+20): subs   w8, w8, #0x1
0x10000030b4 (+24): cset   w8, le
0x10000030b8 (+28): tbnz   w8, #0x0, 0x1000030e4   ; (+72) at main.cpp
0x10000030bc (+32): b      0x1000030c0             ; (+36) at main.cpp:13:14
0x10000030c0 (+36): ldr    w8, [sp, #0x8]
0x10000030c4 (+40): str    w8, [sp, #0x4]
0x10000030c8 (+44): ldr    w8, [sp, #0x8]
0x10000030cc (+48): subs   w0, w8, #0x1
0x10000030d0 (+52): bl     0x10000309c             ; (+0) at main.cpp:11
0x10000030d4 (+56): ldr    w8, [sp, #0x4]
0x10000030d8 (+60): mul    w8, w8, w0
0x10000030dc (+64): stur   w8, [x29, #-0x4]
0x10000030e0 (+68): b      0x1000030f0             ; (+84) at main.cpp:16:1
0x10000030e4 (+72): mov    w8, #0x1
0x10000030e8 (+76): stur   w8, [x29, #-0x4]
0x10000030ec (+80): b      0x1000030f0             ; (+84) at main.cpp:16:1
0x10000030f0 (+84): ldur   w0, [x29, #-0x4]
0x10000030f4 (+88): ldp    x29, x30, [sp, #0x10]
0x10000030f8 (+92): add    sp, sp, #0x20
0x10000030fc (+96): ret
```

ArmV8 - RISC

In most cases, a compiler's ultimate output is the executable code, which is the finite sequences of the instructions that will be fed to the CPU. Each CPU exposes its programmability through its ISA (Instruction Set Architecture), which defines the instructions by groups (such as those arithmetic and logic, status and flow control, data moving, and so on), addressing modes in conjunction with memory and IO management mechanisms, register file, modes of operation (word length, process space size, privilege levels, and so on), properties critical to concurrency control, and more. The ISA is the most critical, major determinant of the translation to be performed by the compiler.

Architecture

```
int f(int p)
{
  if (p>1)
    return p*f(p-1);

  return 1;
}
```

Identical semantics mapped different Instruction Set Architectures (ISA)

ISA can also be defined at software level that may be interpreted or translated further targeting another lower level ISA.

```
15 ssr 1[1 0x00000001]              ; Prologue f 50n(i0ni0n)
16 ssr 29[90 0x0000005a]            ; Debug expression prologue p>1)
17 pmw base pointer offset -3
18 psh int8 1 0x01
19 cvt_i_t -1 regs
20 gre_i
21 czf
22 ssr 3[1 0x00000001]
23 jz 37
24 ssr 29[109 0x0000006d]           ; Debug expression prologue p*f(p-1);
25 pmw base pointer offset -3
26 psh signature entry point 1
27 pmw base pointer offset -4
28 pmw base pointer offset -3
29 psh int8 1 0x01
30 cvt_i_t -1 regs
31 sub_i
32 clf 1
33 ssr 6
34 mul_i
35 ssr 5[1 0x00000001]
36 jmp 41
37 ssr 29[138 0x0000008a]           ; Debug expression prologue 1;
38 psh int8 1 0x01
39 cvt_i_t -1 regs
40 ssr 5[1 0x00000001]
41 ssr 2[1 0x00000001]              ; Epilogue f 50n(i0ni0n)
42 rtf 1
```

A sample stack machine

Instruction Set Architecture can be defined at software level for the purposes of emulation, interpretation, or similar. The target generated for software defined ISA can also be translated further to processor level ISA. The just in time (JIT) compiler that is part of Java execution model is a good example of this two level. According to this model, Java source is translated to bytecodes by the java compiler, then the JIT compiler translates the bytecode to the processor native code to enable execution. You can see the Oak as a historical mark, Dalvik as a virtually modern approach on Android, ART as a target scheme on Android.

Architecture

int f(int p)
{
    if (p>1)
        return p*f(p-1);

    return 1;
}

Identical semantics mapped different Instruction Set Architectures (ISA)

Application Binary Interface (ABI) as OS dependent architectural manifestation.

x64 – Windows ABI

https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170

x64 – Linux ABI

https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/linux-x64-calling-convention-stack-frame

Even if the set instruction set architectures are the most dominant determinant in generation of the back-ends, the software layers that underpin the execution of the programs must also be considered as part of the architecture which the compiler must be conformant to. The application binary interface (ABI) requirements that define the parameter passing conventions can be different between the operating systems even if they run on the same hardware. The 64 bit versions of Windows and Linux use different ABIs so the code generators must be developed keeping the differences in mind even if they run on the same Intel based PCs for example. On top of these, it is quite possible to develop a code generator that uses totally different, custom ABI to run code in an isolated fashion for some application specific reasons. Integer parameters are passed using 4 register fast call on Windows (RCX, RDX, R8, and R9), 6 register fast call on Linux (RDI, RSI, RDX, RCX, R8, R9). There are more conventional differences that have to be comprehended and applied in machine code synthesis.

For more, see https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170 for windows, https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/linux-x64-calling-convention-stack-frame for Linux.

However, the lower end architecture has more to do with from the perspective of a compiler. There may be cases where the hardware elements from external to the CPU must be considered. As a contemporary hot topic, GPU code generation can be given as an example. NVidia has a custom C++ compiler (CUDA C - nvcc) to generate and run GPU kernels. Industry leader companies are in continuous research and development phases aiming at better GPU code generation.

See, https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/ to have a rough idea on a customized language processor.

See the presentation from AMD as a very informative resource, "Nicolai Hähnle, Code Generation for AMD GPUs, 2023, AMD". Source: https://db.in.tum.de/teaching/ws2223/codegen/CodegenForGPUs.pdf?lang=en Retrieved on Feb 2nd, 2024.
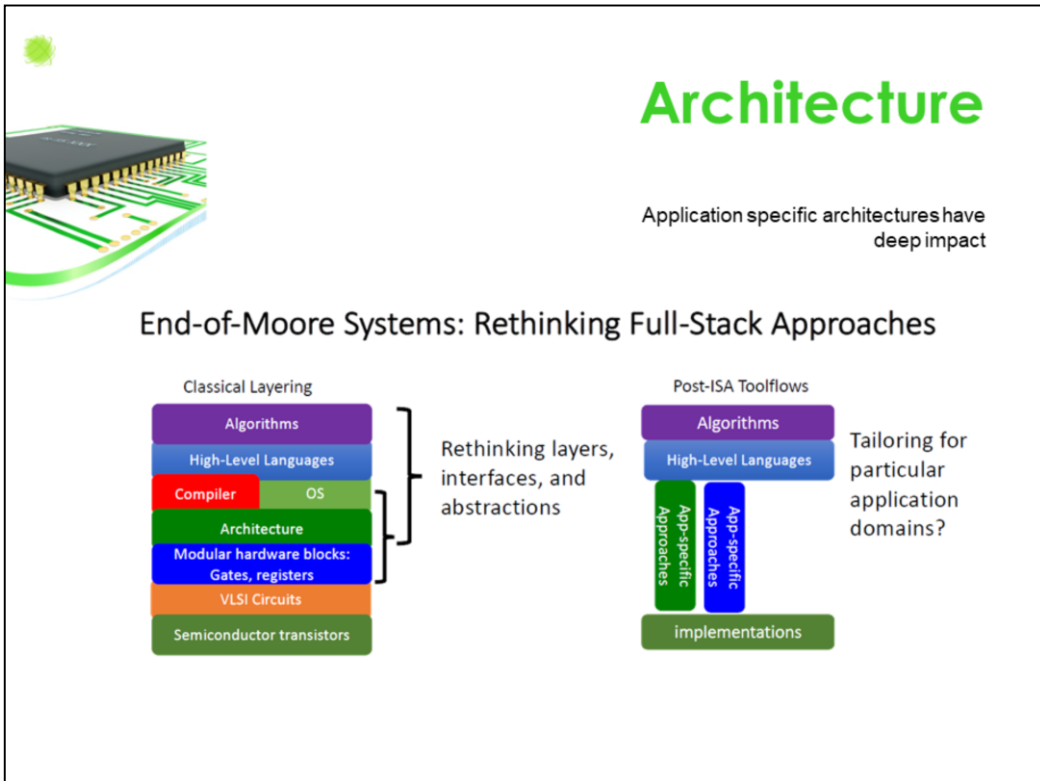
As a final remark on the architecture, we must consider the architectural properties that may span multiple layers of a system. Hardware and lower-level software components may become largely variable and application-specific in a way to force the developers to update multiple the layers of their language processors. The impact of alternatives may be so deep that changes in the language definition and the semantics of the whole translation becomes inevitable, most probably in an enriching fashion.

Lexical analysis is the first stage of language processing that "touches" the input at character level. Recognition of tokens, which are the indivisible elements of the language grammar, happens at the lexical analysis phase. The layer that performs the lexical analysis, the lexical analyzer, while having no idea about the general syntactic rules of the language, generates the vital input to the syntactical analysis phase.

We will focus on common scheme of lexical Analysis and parsing and make some basic definitions including but not limited to prefixes, suffixes, substrings, subsequences, and some related proper forms. Input scanning and regular expressions will be the next topic leading us to the discussions of the non-deterministic and deterministic automatons. We will examine generation of the related structures along with the algorithms that are generators and recognizers.

Finally we will focus on the connection of the lexical analysis to the semantic representation, which is referred to as the symbol table in some resources. At the end, we will go through a few experiments by making use of lexer generators.

The central problem of the syntactic analysis will be language grammars, and parsing strategies. Parse trees, abstract syntax trees, augmentation, analysis, and synthesis of the attributes will be focused on in connection with the semantic representation that evolves as the language processing proceeds. Our vocabulary will be enriched further with the terms like context free grammars, grammar rules, derivations, sentential forms, left and right recursion, ambiguity, parsing tables, look ahead, and more. Each of the terms will come in broader set of contexts that focus on essential knowledge to comprehend the parsing process and probable pitfalls.

Our study on this topic will be finalized with additional experiments by making use of parser generators. Hopefully, these experiments that will be performed at the end of this topic will give the students the sense of creating something tangible at preliminary level.

Quick View 3 / 4

Semantic Analysis

- Meaning and typical patterns
- Type systems
- Attributes, SDD
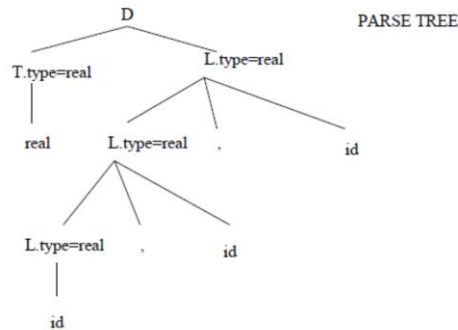- Representing "the meaning"
- Experiment

PARSE TREE

Diagram from CENG 444 course notes, Prof. Dr. Cem Bozşahin

The discussion up until the semantic analysis will be "context free". However, most of the language designs that need to be dealt with will be context dependent or context sensitive. The semantic analysis part is highly dependent on the execution model that will be established in order to implement the "meaning" of the language and the techniques that are chosen by the implementing body or bodies. This means there is no unique way of implementing the meaning. Further to this, since the imperative languages are not the only domain where language processors bring solutions to, our study will be doomed to focus on a few the well-known methods on a few well-known problems. Type system related concepts such as symbol tables, name spaces, scopes, resolution strategies, static and dynamic type checking, type equivalence, type dependence, type arithmetic, and similar will be examined.

Applied track of the course will be extended in a way to cover simplified symbol table management, type scoping, and type arithmetic.

We will start with target architectures, application binary interfaces and calling conventions.

There is so much that can be talked on. We have to limit ourselves to fit the calendar dependent pressures since this is a single semester course. We will cover intermediate representations, simple optimizations, and evaluation of instruction set architectures. Additionally, the challenges of control flow structures, useful topics like data flow analysis, static single assignment forms, register allocation schemes will be mentioned without getting into deep details.

Finally, another extension will be carried out to complete the applied track of the course. At this step you will create components that generate experimental real x64 codes with simplified optimization and register allocation strategies.