



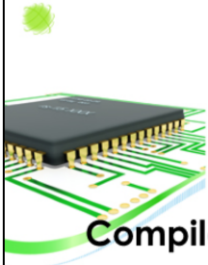
Definition

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.

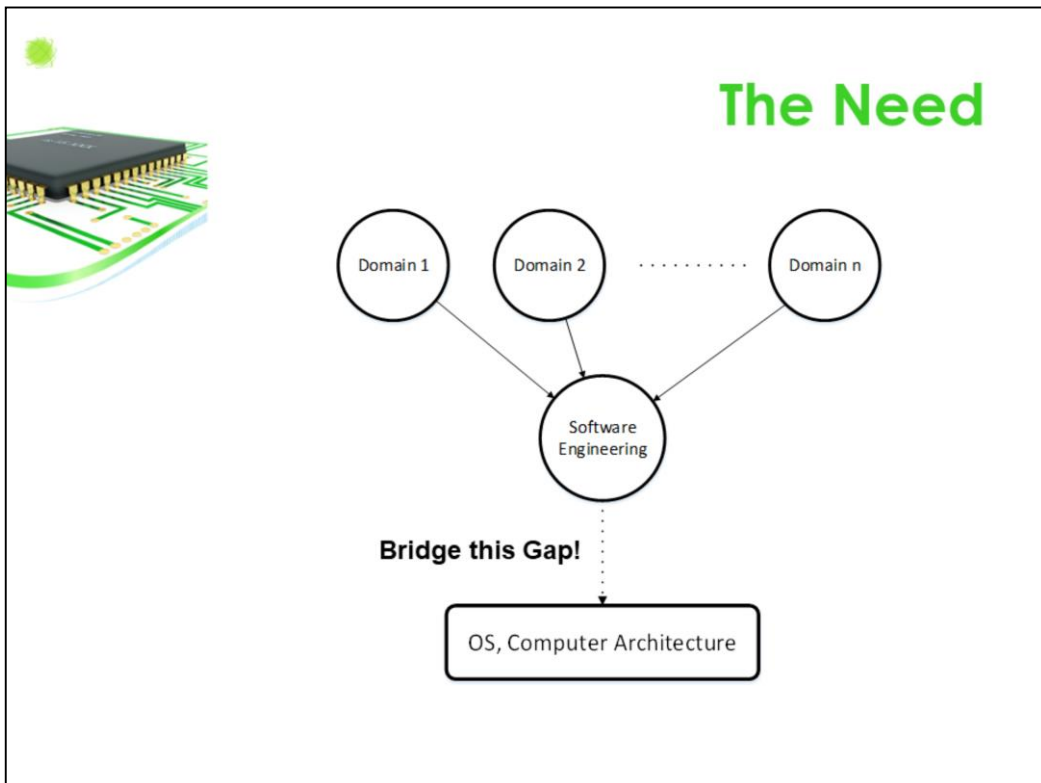
Aho, A.V, Ullman J.D, Sethi R., Lam M.S: Dragon Book

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 than 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.



Yet Another One?

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

Reasons

- Evolutionary
- Technical
- Practical
- Commercial
- ...

**400+ imperative
languages noted so far.**

https://en.wikipedia.org/wiki/Timeline_of_programming_languages
Latest retrieval Feb 2nd, 2024

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

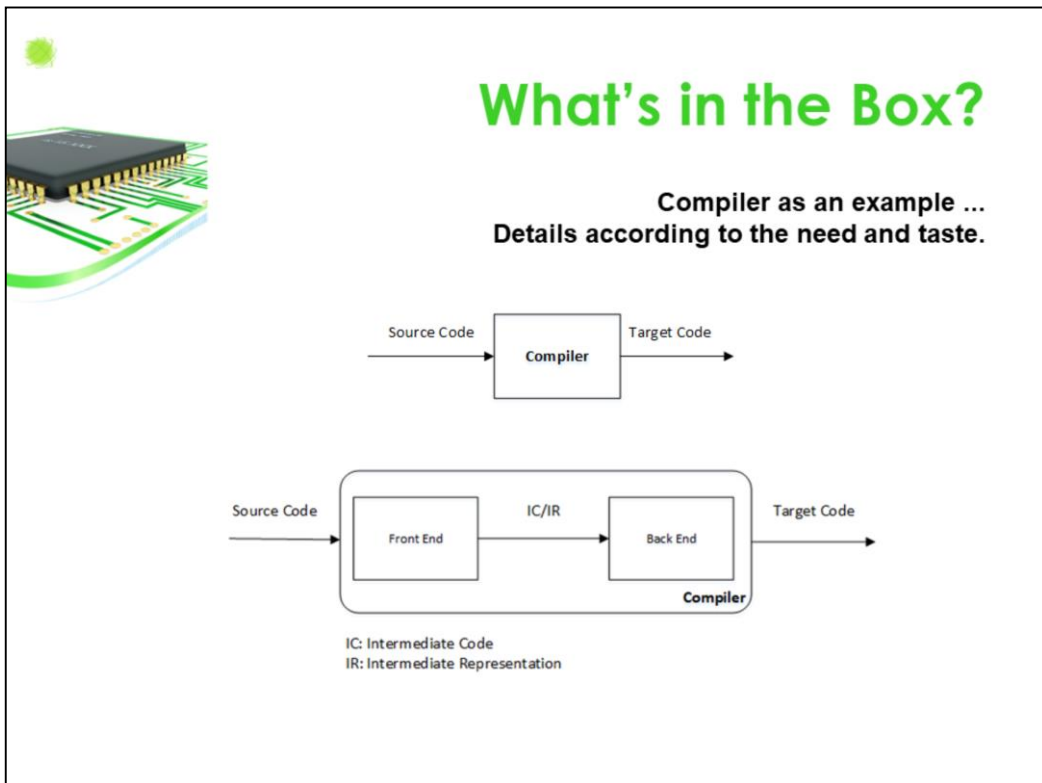
"Much of the past Soviet systems software effort has been in programming languages. This is reflected in the large proportion of the open publications devoted to this area, and is consistent with the given hardware constraints, the relatively formal academic orientation of Soviet software research personnel, 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.

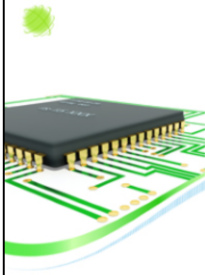
As a last remark, we need to keep in mind that when it comes to deciding to create a new language processors, most probably the reasons will come in combination not in isolation.



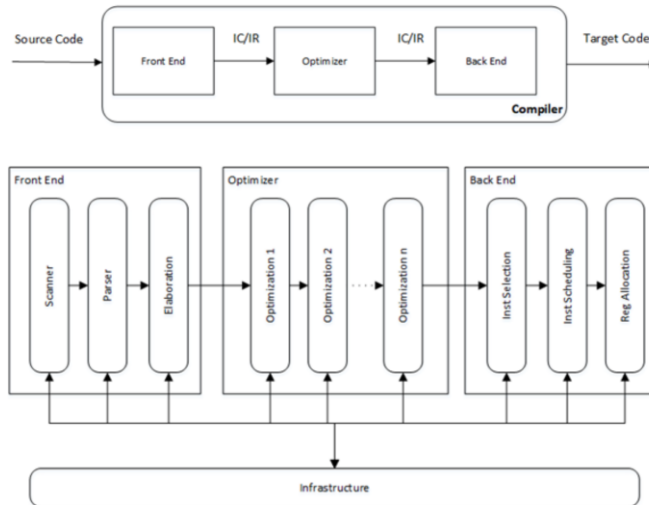
A language processor must have the capacity to check conformance of its input to the formal and the semantic requirements as specified by the language definition. Further to this, it must have the capacity to perform all necessary translations to synthesize target representations and / or to perform 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.

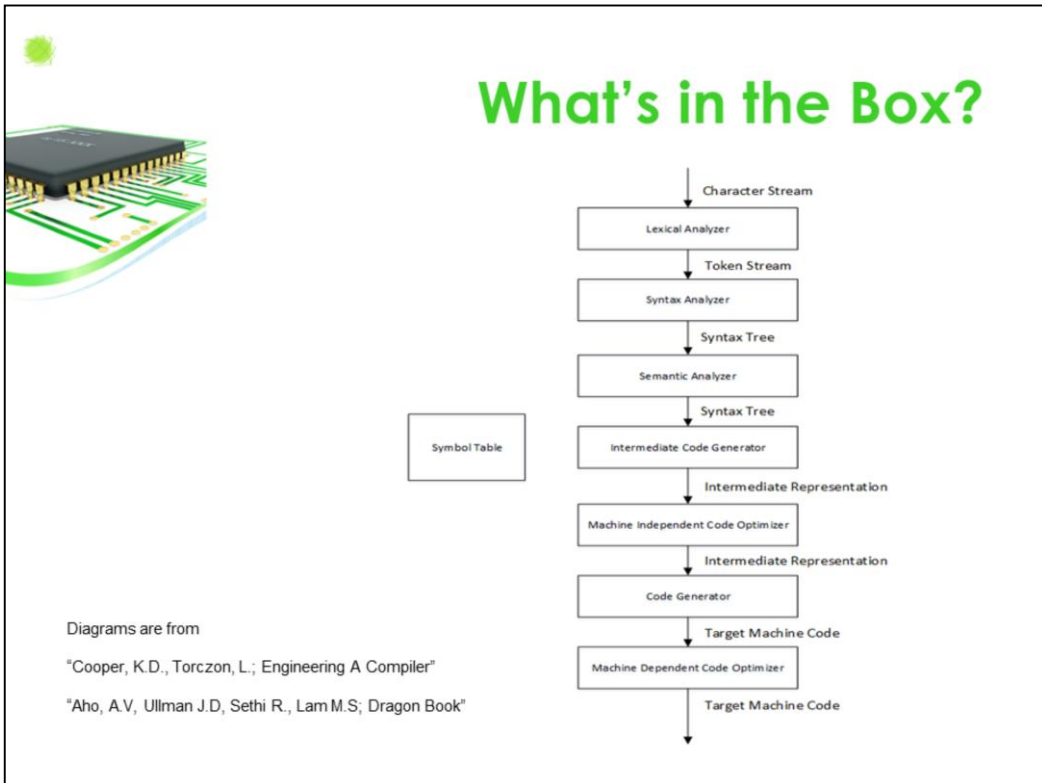
Let's see a few "box models" that are valid for the compilers, which are the language processors we will focus in this course. In the first model, the compiler is a single magic box that receives the source code and generates the target code. Following models are developed by the authors to convey a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

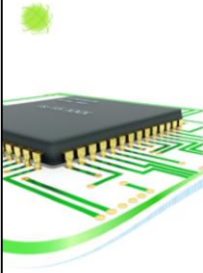


What's in the Box?





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

<pre>000077ff79ef52320 09 4c 24 00 mov dword ptr [rsp+8],ecx 000077ff79ef52324 55 push rbp 000077ff79ef52325 57 push rdi 000077ff79ef52326 48 b1 cc e8 00 00 00 sub rbp,0f0h 000077ff79ef52320 48 b0 6c 24 20 lea rbp,[rsp+20h] 000077ff79ef52332 48 bd 00 f0 0c 01 00 lea rcx,[_f791f1de_lecture01@cpp (07ff79ef63029h)] 000077ff79ef52339 e8 aa f0 ff ff call __CheckForDebuggerJustlyCode (07ff79ef5136h) 000077ff79ef5233e 5f (p25) 000077ff79ef5233e 83 bd e8 00 00 00 01 cmp dword ptr [p],1 000077ff79ef52345 7e 1c jle f443h (07ff79ef52363h) 000077ff79ef52347 05 05 e8 00 00 00 mov eax,dword ptr [p] 000077ff79ef5234d ff c8 dec eax 000077ff79ef5234f 08 c8 mov ecx,eax 000077ff79ef52351 e8 75 ff ff ff call f (07ff79ef512cbh) 000077ff79ef52356 b8 bd e8 00 00 00 mov ecx,dword ptr [p] 000077ff79ef5235c 0f af c8 imul ecx,eax 000077ff79ef5235f 08 c1 mov eax,ecx 000077ff79ef52361 e8 05 jmp f440h (07ff79ef52363h) 000077ff79ef52363 05 01 00 00 00 mov eax,1 000077ff79ef52368 48 bd a5 c8 00 00 00 lea rdi,[rbp+0cbh] 000077ff79ef5236f 5f pop rdi 000077ff79ef52370 5d pop rbp 000077ff79ef52371 c3 ret</pre>	<pre>0x10000309c <40>: sub sp, sp, #0x20 0x1000030a0 <44>: stp x29, x30, [sp, #0x10] 0x1000030a4 <48>: add x29, sp, #0x10 0x1000030a8 <12>: str w0, [sp, #0x8] 0x1000030ac <16>: ldr w0, [sp, #0x8] 0x1000030b0 <20>: sub w0, w0, #0x1 0x1000030b4 <24>: cset w0, le 0x1000030b8 <28>: tbz w0, #0x0, 0x1000030e4 ; <72> at main.cpp 0x1000030bc <32>: b 0x1000030c0 ; <36> at main.cpp:13:14 0x1000030c0 <36>: ldr w0, [sp, #0x8] 0x1000030c4 <40>: str w0, [sp, #0x4] 0x1000030c8 <44>: ldr w0, [sp, #0x8] 0x1000030cc <48>: sub w0, w0, #0x1 0x1000030d0 <52>: bl 0x10000309c ; <40> at main.cpp:11 0x1000030d4 <56>: ldr w0, [sp, #0x4] 0x1000030d8 <60>: mul w0, w0, w0 0x1000030dc <64>: stur w0, [x29, #0x4] 0x1000030e0 <68>: b 0x1000030f0 ; <84> at main.cpp:16:1 0x1000030e4 <72>: mov w0, #0x1 0x1000030e8 <76>: stur w0, [x29, #0x4] 0x1000030ec <80>: b 0x1000030f0 ; <84> at main.cpp:16:1 0x1000030f0 <84>: ldr w0, [x29, #0x4] 0x1000030f4 <88>: ldp x29, x30, [sp, #0x10] 0x1000030f8 <92>: add sp, sp, #0x20 0x1000030fc <96>: ret</pre>
--	--

x64 - CISC

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

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.

```

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

    return 1;
}

```

```

15 ssr 1[1 0x00000001] ; Prologue f 50n(10n10n)
16 ssr 29[90 0x0000002a] ; Debug expression prologue p>1
17 pmw base pointer offset -3
18 psh int8 1 0x01
19 cvt_i_t -1 regs
20 gre_1
21 clif
22 ssr 3[1 0x00000001]
23 jz 37
24 ssr 29[109 0x0000000d] ; 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 1
32 clif 1
33 ssr 6
34 mul 1
35 ssr 5[1 0x00000001]
36 jmp 41
37 ssr 29[138 0x0000000a] ; 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(10n10n)
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

Identical semantics mapped different Instruction Set Architectures (ISA)

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

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

    return 1;
}
```

```
000077ff79ef52320 89 4c 24 00      mov     dword ptr [rsp+0],ecx
000077ff79ef52324 55              push    rbp
000077ff79ef52325 57              push    rdi
000077ff79ef52326 48 b1 ec 00 00 00 sub     rsp,000h
000077ff79ef52320 48 b0 0c 24 20    lea     rbp,[rsp+20h]
000077ff79ef52322 48 d0 00 f0 0c 01 00 lea     rcx,[_F79f1DE_lecture01@cpp (07ff79ef53029h)]
000077ff79ef52329 e8 aa f0 ff ff    call    _CheckForDebuggerJustlyCode (07ff79ef5330eh)
000077ff79ef5233e      if (p>1)
000077ff79ef5233e 83 80 e0 00 00 01 cmp     dword ptr [p],1
000077ff79ef52345 7c 1c           jle     f443h (07ff79ef52363h)
000077ff79ef52347 8b 05 e0 00 00 00 mov     eax,dword ptr [p]
000077ff79ef52347 8b 05 e0 00 00 00 mov     ecx,ecx
000077ff79ef5234f 8b c8           dec     ecx,ecx
000077ff79ef52351 e8 75 ff ff ff    call    f (07ff79ef5120bh)
000077ff79ef52356 48 d0 00 00 00 00 mov     rcx,dword ptr [p]
000077ff79ef5235c 0f af c8        imul    ecx,ecx
000077ff79ef5235f 8b c1           mov     eax,ecx
000077ff79ef52361 e8 05           jmp     f440h (07ff79ef52360h)
000077ff79ef52363 8b 01 e0 00 00 00 mov     eax,1
000077ff79ef52368 48 bd a5 c8 00 00 00 lea     rsp,[rbp+0c0h]
000077ff79ef5236f 5f              pop     rdi
000077ff79ef52370 5d              pop     rbp
000077ff79ef52371 c3              ret
```

x64 – Windows ABI

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

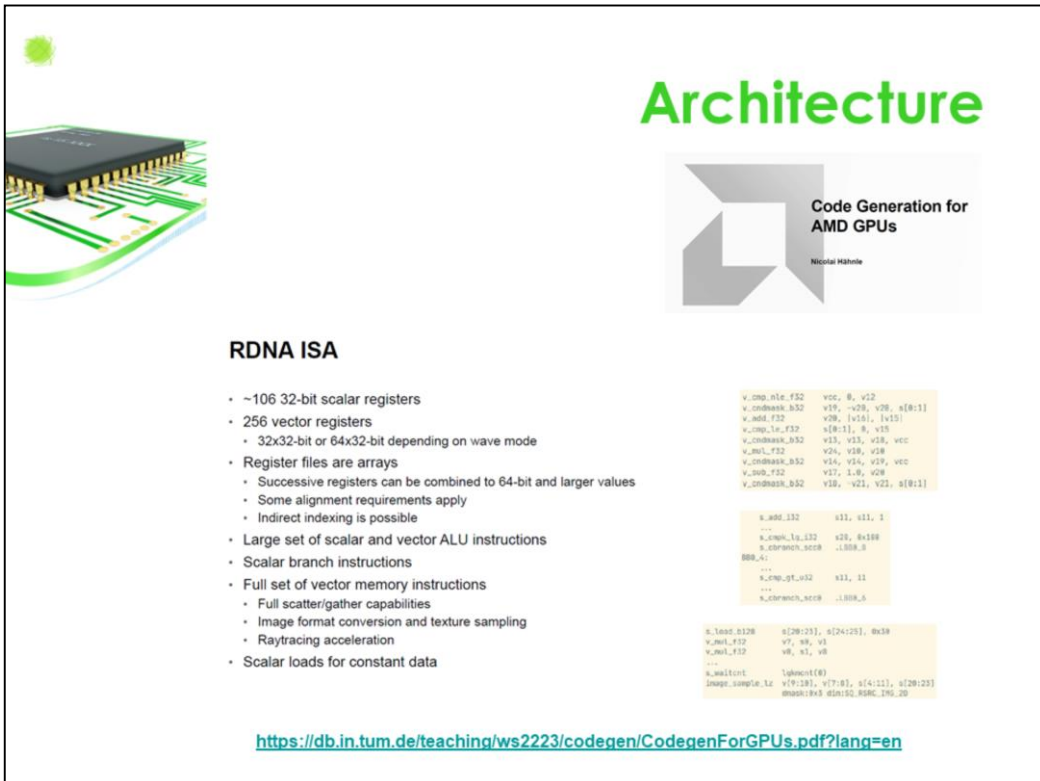
```
00005555555551cd: push    %rbp
00005555555551ce: mov     %rsp,%rbp
00005555555551d1: sub     $0x10,%rsp
00005555555551d5: mov     %edi,-0x4(%rbp)
14      if (p>1)
00005555555551d8: cmpl    $0x1,-0x4(%rbp)
00005555555551dc: jle     0x5555555551f1 <_Z1fi+40>
15      return p*f(p-1);
00005555555551de: mov     -0x4(%rbp),%eax
00005555555551e1: sub     $0x1,%eax
00005555555551e4: mov     %eax,%edi
00005555555551e6: call    0x5555555551c9 <_Z1fi>
00005555555551eb: imul    -0x4(%rbp),%eax
00005555555551ef: jmp     0x5555555551f6 <_Z1fi+45>
17      return 1;
00005555555551f1: mov     $0x1,%eax
00005555555551f6: leave   %eax
00005555555551f7: ret
```

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.



The slide is titled "Architecture" in large green letters. On the left, there is a 3D illustration of a GPU chip on a circuit board. On the right, there is a logo for "Code Generation for AMD GPUs" by "Nicolai Hähle". The main content is a list of features for the RDNA ISA, followed by three blocks of assembly code examples.

Architecture

RDNA ISA

- ~106 32-bit scalar registers
- 256 vector registers
 - 32x32-bit or 64x32-bit depending on wave mode
- Register files are arrays
 - Successive registers can be combined to 64-bit and larger values
 - Some alignment requirements apply
 - Indirect indexing is possible
- Large set of scalar and vector ALU instructions
- Scalar branch instructions
- Full set of vector memory instructions
 - Full scatter/gather capabilities
 - Image format conversion and texture sampling
 - Raytracing acceleration
- Scalar loads for constant data

```

v_cmp_nle_f32 vcc, R, v12
v_cdbash_b32 v19, v28, v29, s[R:1]
v_add_f32 v29, [v16], [v15]
v_cmp_le_f32 s[R:1], R, v15
v_cdbash_b32 v13, v13, v18, vcc
v_mul_f32 v24, v18, v18
v_cdbash_b32 v14, v14, v19, vcc
v_sub_f32 v17, v16, v08
v_cdbash_b32 v18, v21, v21, s[R:1]

s_wrt_132 s11, s11, 1
...
s_cmphl_1q_132 s28, s108
s_branch_scc8 .LBBR_5
BBR_4:
...
s_cmp_gt_v32 s11, s11
...
s_branch_scc8 .LBBR_5

s_load_b128 s[20:23], s[24:25], s38
v_mul_f32 v7, s9, v1
v_mul_f32 v8, s1, v8
...
s_waitcnt lgprcnt(8)
smnpr_sample_1z v[7:8], v[7:8], s[4:11], s[20:23]
smnpr[8x8] smnpr[8x8] smnpr[8x8]

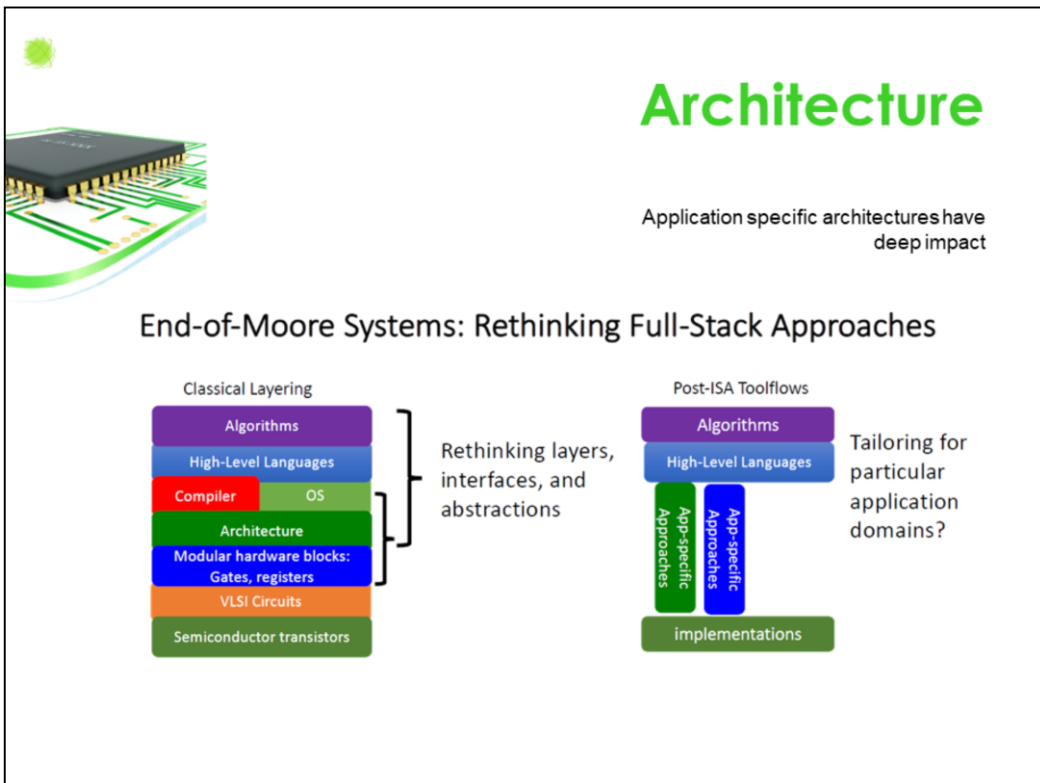
```

<https://db.in.tum.de/teaching/ws2223/codegen/CodegenForGPUs.pdf?lang=en>

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 Hähle, 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.



Quick View 1 / 4

Lexical Analysis

- Reserved words, tokens
- Regular expressions
- NFA, DFA structures
- Optimized structures, strategies
- Generators and recognizers
- Experiment

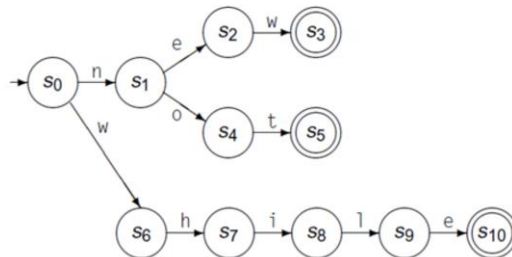


Diagram from "Cooper, K.D., Torczon, L.; Engineering A Compiler"

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 syntactic 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 automaton. 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.



Quick View 2 / 4

Syntactic Analysis

- Context Free Grammars, BNF
- Derivations, sentential forms
- Parse trees, AST
- Parsing strategies
- Parsing algorithms
- Experiment

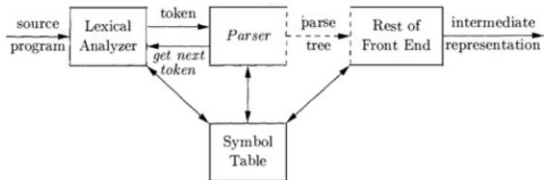


Diagram from "Aho, A.V., Ullman J.D., Sethi R., Lam M.S.; Dragon Book"

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

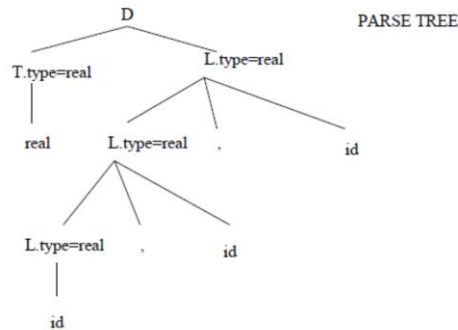


Diagram from CENG 444 course notes, Prof. Dr. Cem Bozşahin

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.



Quick View 4 / 4

Towards Back End Processing

- Target architectures
- ABIs, calling conventions
- Intermediate code
- Optimizations
- Register allocation
- Compiler runtime
- Finally, writing target!
- Experiment

Synthesis

Code Generation

Back End

```
push    %rbp
mov     %rsp, %rbp
sub     $0x10, %rsp
mov     %edi, -0x4(%rbp)
cmpl    $0x1, -0x4(%rbp)
jle     0x555555551f1 <_Z1fi+40>
mov     -0x4(%rbp), %eax
sub     $0x1, %eax
mov     %eax, %edi
call    0x555555551c9 <_Z1fi>
imul    -0x4(%rbp), %eax
jmp     0x555555551f6 <_Z1fi+45>
mov     $0x1, %eax
leave
ret
```

... and a few words on

- Data flow analysis
- Static single assignment forms
- Instruction scheduling

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.