

Almost every compiler has 5 interfaces

1. lexical Analysis
2. parsing
3. Semantic Analysis → this is hard
such as variable binding
4. optimization
5. code generation

The proportions have changed since Fortran.



Economy of Programming language

1. why so many programming language .

scientific computing { good FP
good arrays
parallelism } FORTRAN formula translation

system programming { control of resources
real-time constraints }

C/C++

2. Why are there new programming language

programmer training is the dominant cost for a programming language.

- [1. widely-used languages are slow to change.
- 2. Easy to start a new language.
- 3. languages adopted to fill a void.

New programming language tends to look like old languages.

3. What is a good programming language?

There is no universally accepted metric for language design.

Formal language Note: this section belongs to parsing phrase

Definition

Let Σ be a set of characters (an alphabet). A language over Σ is a set of strings of characters drawn from Σ .

Examples

Alphabet: ASCII

Language: C programs

Meaning functions L maps syntax to semantics.

L is the meaning function
beneficial for notations

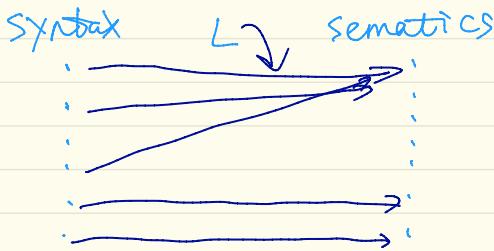
$$\left[\begin{array}{lcl} L(\Sigma) & = & \{ \text{" "} \} \\ L('c') & = & \{ \text{"c"} \} \\ L(A+B) & = & L(A) \cup L(B) \\ L(AB) & = & \{ ab \mid a \in L(A), b \in L(B) \} \\ L(A^*) & = & \bigcup_{i \geq 0} L(A)^i \end{array} \right]$$

semantics \rightarrow set operation

Why use a meaning function?

1. Make clear what is syntax and what is semantics
2. Allow us to consider notations as a separate issue
3. Because expressions and meanings are not 1-1
multiple ways to write an expression that means the same thing

Don't under estimate the importance of notations



Meaning is many to one
Never is one-to-many!

into to parsing

Input : sequence of tokens from lexer
Output : parse tree of the program

phrase
lexer
parser

Input Output
string of characters string of tokens
 parse tree
string of tokens

may be implicit

→ compiler may combine these two
phrases into one. All these things are
done by a parser.

Context Free Grammars CFGs

↳ This section belongs to parsing phrase

△ Not all strings are programs.

- 1. a language to describe valid string of tokens
- 2. a method to distinguish valid from invalid string of tokens.

△ programming language have recursive structure.

An EXPR is

- if EXPR Then EXPR else EXPR fi
- while EXPR loop EXPR posl
- ...
- ...

Context-Free grammars are a natural notation for this recursive structure.

A CFG consists of

- A set of terminals of T
- A set of non-terminals N
- A start symbol S (SEN)
- A set of productions

$$X \rightarrow Y_1 \dots Y_N$$

a set of replacement rules $X \in N$

$$Y_i \in NT \cup \{\epsilon\}$$

$$\rightarrow \left\{ \begin{array}{l} S \rightarrow (S) \\ S \rightarrow \epsilon \end{array} \right\} \quad N = \{S\}$$

$$T = \{(,)\}$$

productions can be regarded as replacement rules.

1. Begin with the string with the only start symbol S
2. Replace with any non-terminal X in the string by the right-hand side of some production $X \rightarrow Y_1 \dots n$
3. Repeat (2) until there are no non-terminals

The language for CFG.

Let G be a context-free grammar with start symbol S . Then the language $L(G)$ of G is:

$$\underbrace{\{a_1 \dots a_n \mid \text{there exists } i \in T \text{ such that } S \xrightarrow{*} a_i \dots a_n\}}$$

the language is
a series of symbols

All the strings of terminals
can be derived by beginning with
just a start symbol

Summary:

1. No rules for replacing terminals.
2. Once generated, terminals are permanent.
3. Terminals ought to be tokens of the language.

SKIP several details of parsing now.

Semantic Analysis

- △ lexical Analysis
Detects inputs with illegal tokens
- △ parsing
Detects inputs with ill-formed parsed trees
- △ semantic parsing
"last" front-end phrase.
Catches all remaining errors

context-free
parsing ?

what does
"context" matter?

Why a separated semantic parsing interface

1. parsing cannot catch some errors
2. Some language constructs are not context-free. how to understand this?

some checks :

1. all identifiers are declared
2. types

Scope

Matching identifier declarations with use.

Important static analysis in most languages.

1. static scope
most languages
2. dynamic scope
a few languages

Scope depends on execution of the program

```
let X: Int <- 0 in
{
    X;
    let X: Int <- 1 in
        X;
    X;
}
```

A dynamically-scoped variable refers to the closest enclosing binding in the execution of the program.

Types The goal of type checking is to ensure that operations are used **ONLY** with the correct type.

Consensus

1. a set of values
2. A set of operations on those values

classes

are one instantiation of the modern notion of type.

1.

statically typed: All or almost all checking of types is done as part of compilation.
C / C++ / Java

2. dynamically typed: Almost all checking of types is done as part of program execution.
Lisp / Python / Perl

3. untyped : machine code

static vs. dynamic typing

static typing ① catches many programming errors at compile time.

② Avoid overhead of runtime type checking

dynamic typing

① static typing systems are restrictive

② rapid prototyping difficult within a static typing type system.

static typing vs. dynamic typing

1. A lot of code is written in statically typed language with an "escape" mechanism.
 - ✓ unsafe cast in C++ / Java
2. People retrofit static typing to dynamically typed language.
3. It is debatable whether either compromise represents the best or worst of both worlds.

The compiler infers types for expressions.

the process of verifying fully-typed programs.

the process of filling in missing type information!

These two are different, but the two terms are often used interchangeable.

Type checking

Type inference

Semantic Analysis → Symbol Tables

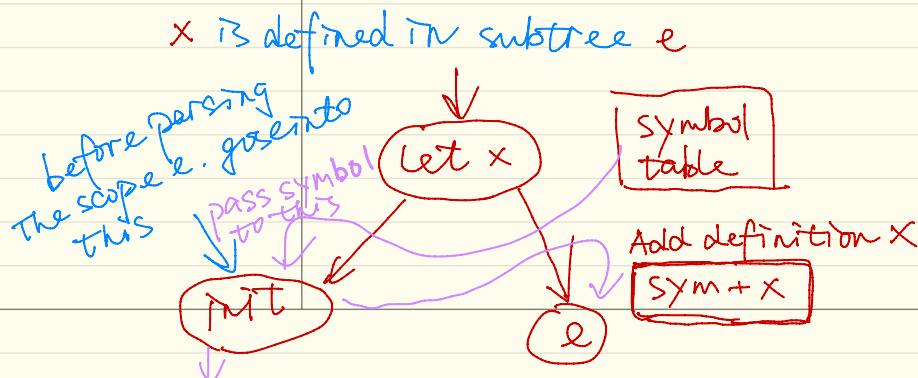
A data structure maintained by the compiler that tracks the current bindings of identifiers.

Much of semantic analysis can be expressed as a recursive descent of an AST algorithms.

- Before: Process an AST node n
- Recurse: Process the children of n
- After: Finishing processing the AST node n

When performing semantic analysis on a portion of the AST, we need to know which identifiers are defined.

Example: $\text{let } x: \text{Int} \leftarrow 0 \text{ in } e$



something like
type checking

let $x: \text{Int} \leftarrow 0$ in e

1. before processing e, add definition of x into current definitions, overriding any other definitions of x
2. Recurse
3. After processing e, remove definition of x , and restore old definition of x .

We can use a stack to implement a simple symbol table.

A simple interface of a symbol table manager

1. enter_scope() start a new nested scope
2. find_symbol(x) finds current x (or null)
3. add_symbol(x) add a symbol x to the table
4. check_scope(x) true if x defined in current scope
5. exit_scope() exit current scope

Semantic parsing requires multiple passes.

not very clear about this

static vs. dynamic Type System

static types detect common errors at the compile time.

but some correct programs are disallowed.

1. some argue for dynamic type checking
2. others want more expressive static type checking.

→ But more complex

dynamic type:

① a runtime notion
static type of an expression captures all dynamic types the expression could have.

② a compile-time notion

class A { }

class B inherits A { ... }

class Main {

x : A ← new A ; ← dynamic type of x
= B A

x ← new B ; ← dynamic type of x
= B B

static type of
x is A, the
compiler knows
it, and it is
invariant.

In all the scope. x is typed with A
by the compiler, but at runtime, x can take different
runtime types.

soundness theory: for all expression E

$$\forall E, \text{dynamic-type}(E) \leq \text{static-type}(E)$$

- △ All operations that can be used on an object of type C can also be used on an object of type $C' \leq C$

Routine Organization

Recap:

enforce language semantic

the front-end phrase ① enforce language definition

{ lexical analysis ② build the data structures

 | parsing

 | semantic analysis

needed for the code generation
algorithms.

back-end phrase

{ optimization

 | code generation

← now goes into this
will cover three problems.

1. Runtime resource management

2. Correspondence between

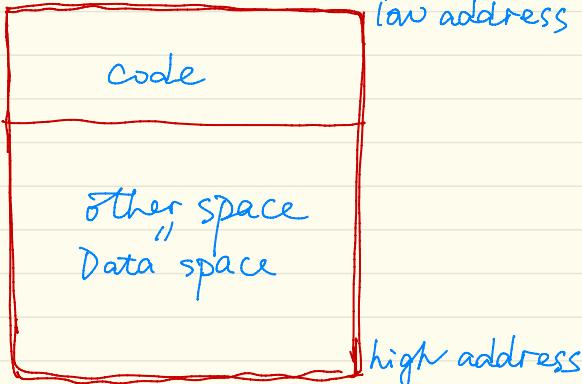
3. storage organization

{ compile-time
 | runtime

Execution of a program is initially under the control of operating system, when a program is invoked:

1. The OS allocates space for the program
2. The code is loaded into part of the space
3. The OS jumps to the entry point (i.e. "main")

Memory



△ This is a simplified picture, not all memory need be contiguous

Compiler is responsible for

1. Generating code.
2. Orchestrating use of the data space.

Activation Records

The information needed to manage one procedure activation is called an activation record (AR) or a frame.

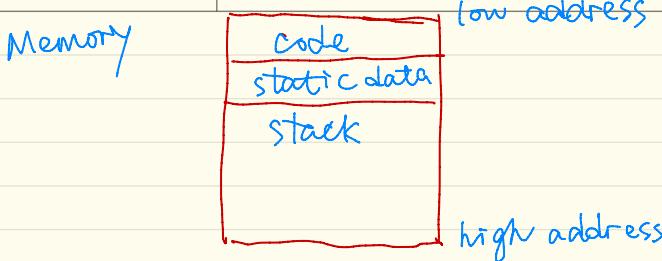
If procedure F calls G , then F 's AR contains a mix of info about G

Why we need AR? \rightarrow Correctly execute procedures.

- $\triangle F$ is "suspended" until G completes at which point F resumes.
- $\triangle G$'s AR contains information needed to
 1. complete execution of G
 2. Resume execution of F

Globals and heap

1. All reference to a global variable point to the same data
 - \triangle can't store a global in an AR
2. Globals are assigned a fixed address once.



3. A value that outlives the procedure that creates it cannot be kept in AR
4. Languages with dynamically allocated data use heap to store dynamic data

Runtime organization

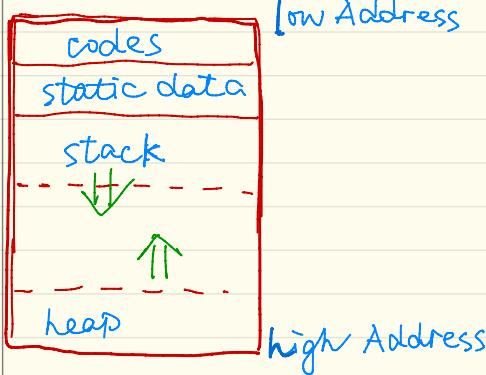
1. The code area contains object code
 - △ for many languages, fixed size and read only
2. The static area contains data (not code) with fixed address.
 - △ Fixed size, may be readable or writable.
3. The stack contains an AR for each currently active procedure.
 - △ Each AR usually fixed size, contains locals
4. Heap contains all other data.

Both the heap and stack grow. Must take care that they don't grow onto each other.



Start heap and stack at opposite ends of memory, and let them grow towards each other.

Memory



Introduction to Code Generation

✓ This lesson focus on generation codes for a stack machine. The codes will run on a real architecture.

- ① The accumulator is kept in MIPS register \$a0
- ② The stack is kept in memory.
The address of the next location on the stack is
- ③ kept in MIPS register \$sp

✓ MIPS architecture

- ① prototypical Reduced Instruction Set Computer (RISC)
- ② Most operations use registers for operands & results
- ③ Use load & store instructions to use values in memory
- ④ 32 general purpose registers (32 bit each)
 - use \$sp \$a0 and \$t1

lw reg, offset(reg₂)

Load 32-bit word from address reg₂ + offset into reg.

add reg₁, reg₂, reg₃
reg₁ ← reg₂ + reg₃

sw reg, offset (reg₂)

store 32-bit word in reg, at address reg₂ + offset

addiu reg₁, reg₂, imm
reg₁ ← reg₂ + imm

li reg imm
reg ← imm

"u" means overflow is not checked

Generate codes for high-level language.
for each expression e , the code generation algorithm:

- ① Compute the value of e in \$as
- ② Preserves \$sp and contexts of stack.

$cgen(e) \rightarrow$ produces a program



i is a constant

$$cgen(i) = li \$a0 i$$

RED

compile time

BLUE

Runtime



$cgen(e_1 + e_2) \rightarrow cgen(e_1)$

sw \$a0 0(\$sp)

addiu \$sp \$sp - 4

$cgen(e_2)$

lw \$t1 4(\$sp)

add \$a0 \$t1 \$a0

addiu \$sp \$sp - 4

highlight some points about code generation

- ① The code for $+$ is a template with "holes" for code for evaluating e_1 and e_2
- ② stack machine code generation is recursive
 - code for $e_1 + e_2$ is code for e_1 and e_2 glued together
- ③ code generation can be written as a recursive descent of the AST

code generation(11)

code for function calls and function definitions depends on the layout of the AR activation record

The activation record must be designed together with the code generator.

Production compiler

1. emphasis is on keeping values in registers
2. Intermediate results are laid out in the AR.
not pushed and popped from the stack.