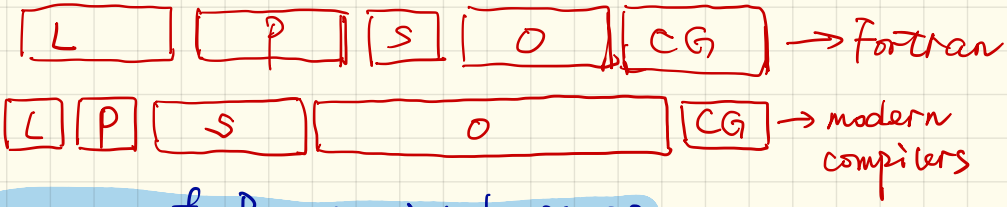
Almost every compiler has 5 interfaces

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.

Semantic Analysis

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

Why a seperated semantic parsing interface

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

context-free parsing ?

what does "context" matter?

some checks:
1. all identifiers are declared
2. types

Scope

Matching identifier declarations with use.

important static analysis in most language.

1. static scope — Scope only depends on program texts not run-time behaviors.

most languages

2. dynamic scope — Scope depends on execution of the program

a few languages

```
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.
   - ✓ unsafed 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.

Type checking the process of verifying fully-typed programs.

Type inference the process of filling in missing type information.

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

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 $\mathcal{L}$ maps syntax to semantics.

$\mathcal{L}$ is the meaning function

$$
\begin{aligned}
\mathcal{L}(\varepsilon) &= \{\text{""}\} \\
\mathcal{L}(\text{'c'}) &= \{\text{"c"}\} \\
\mathcal{L}(A+B) &= \mathcal{L}(A) \cup \mathcal{L}(B) \\
\mathcal{L}(AB) &= \{ ab \mid a \in \mathcal{L}(A),\ b \in \mathcal{L}(B) \} \\
\mathcal{L}(A^*) &= \bigcup_{i \geq 0} \mathcal{L}(A^i)
\end{aligned}
$$

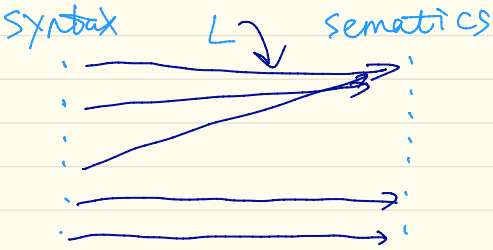
semantics → set operation

beneficial for notations

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 way to write an expression that means the same thing

Don't underestimate 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

	Input	Output
lexer	string of characters	string of tokens
parser	string of tokens	parse tree (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 pool
- ...
- ...

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 ($S \in N$)
- A set of productions — $X \rightarrow Y_1 \cdots Y_N$

a set of replacement rules

$X \in N$

$Y_i \in N \cup T \cup \{\varepsilon\}$

$\{ S \to (S) \}$ $N = \{S\}$

$\{ S \to \varepsilon \}$ $T = \{(,)\}$

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 $X \to Y_1 \cdots Y_n$
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:

$$\{a_1 \ldots a_n \mid \forall i\ a \in T \wedge S \xrightarrow{*} a_i \ldots a_n\}$$

the language is a series of symbols

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

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 → Symbol Tables

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

↳ and also many code generation algorithms.

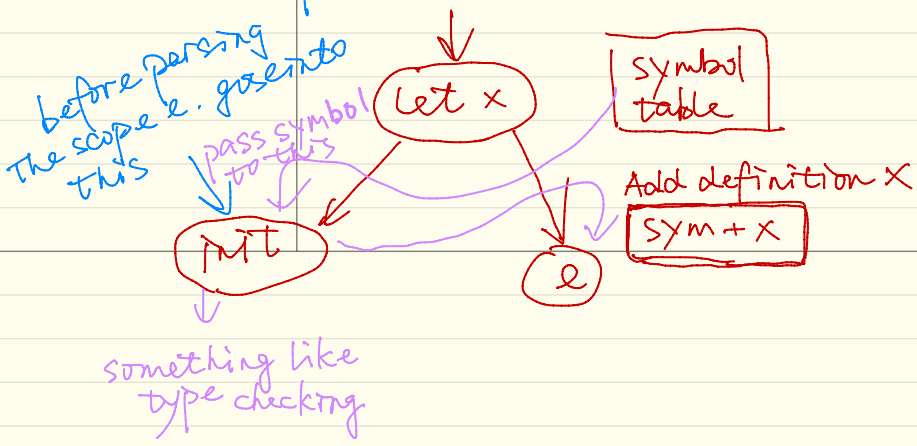
Much of semantic analysis can be expressed as a recursive descent of an AST

- 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: let x:Int ← 0 in e

x is defined in subtree e



let x:Int ← 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 ;
    x ← new B ;
}
```

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

← dynamic type of x is A

← dynamic type of x is B

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

$\forall E$, dynamic_type(E) $\leq$ static_type(E)

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

Runtime Organization

Recap:

the front-end phrase
- lexical analysis
- parsing
- semantic analysis

① enforce language definition
② build the data structures needed for the code generation algorithms.

enforce language semantic

back-end phrase
- optimization
- code generation

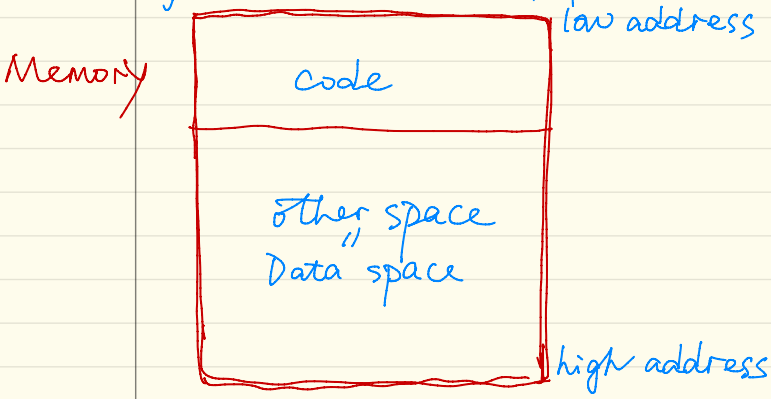
← now goes into this

will cover three problems.

1. runtime resource managment
2. correspondence between { compile-time / runtime }
3. storage organization

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")



△ This is a simplied 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 of F calls G, then F's AR contains a mix of info about G

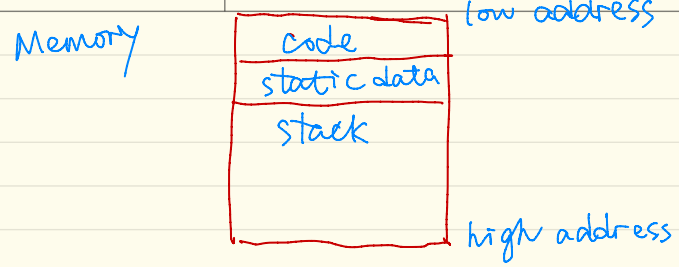
Why we need AR? → Correctly execute procedures.

△ F is "suspended" until G completes at which point F resumes.

△ 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
   △ 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 into each other.

↓↓

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

Memory