Implementation of Programming Languages

Slide 1

→ various ways to bridge this gap:
  - Compiler
  - Interpreter
  - Hybrid approaches

Compiler

- User Source Language
- Compiler
- Machine Code Computer

Compiler translates source language into machine code

Interpreter

- User Source Language
- Compiler
- Machine Code Computer

Interpreter as an abstract machine on top of the concrete machine, executes the source language instructions.

Hybrid Approach

- User Programming Language
- Compiler
- Interpreter
- Machine Code Computer

→ Compiler translates source language into some form of intermediate code, (byte code in case of Java, Elisp)
→ Interpreter as an abstract machine on top of the concrete machine, executes the instructions of the intermediate code.
Compiler Overview

High-level languages:

- no one-to-one correspondence between source code and machine code instructions
- compiler has to "understand" the meaning of the program to a certain extent to be able to translate it

Example Program: C function declaration

```c
int foo () {
    int i;
    i = 11;
    if (i > 5) {
        i = i-1;
    } else {
        i = i+1;
    }
}
```

Lexer

Decomposes string into sequence of tokens

```
"int foo() {
 int i;...
```

Parser

- Lexer produces an unstructured sequence of tokens
- Parser analyses the structure of the program and builds a parse tree
- Correct structure specified by grammar

Comments, spaces, newlines are discarded
Languagespecific
How would you implement a data type Token in C, Java, or Haskell?
Example: grammar for simplified C variant, rule for function definition

► a function definition consists of
  • an identifier (type of return value), followed by
  • an identifier (function name), followed by
  • list of arguments, enclosed in parenthesis, and a
  • statement (function body)

too imprecise, too verbose for the definition of real languages

Grammar of a language usually given as EBNF (Extended Backus-Naur Form)

```
funDef ::= Ident1 Ident2 (arguments) stmt
stmt ::= expr | if expr then stmt | else stmt2 | return expr |
        { locDec stmts } | while ( expr ) stmt
stmts ::= ε | stmt stmts
expr ::= Num | Ident | expr1 + expr2 | expr1 - expr2 |
      | Ident * expr | Ident / expr
locDec ::= Ident1 Ident2 ;
arguments ::= ε | ...
```

What would be a suitable data type definition for a parse tree?

Semantic Analysis

► check (static) semantic properties, for example:
  - identifier defined before use?
  - is the program type correct?
  - symbol table (often hash table) to store information during this phase

Parser

Optimisation and Code Generation
Most compilers implement some kind of optimisations, often, intermediate code is generated for this purpose.

Intermediate code is simpler, closer to assembly language, sometimes in effect functional (SSA).

Inlining: Calls to simple functions can sometimes be replaced with the function body in the code:

Before inlining:
```c
int pred(int x) { if (x == 0) return 0; else return x - 1; }
int f(int y) {
    return pred(y) + pred(0) + pred(y+1);}
```

After inlining:
```c
int f(int y) {
    int temp = 0;
    if (y == 0) temp += 0; else temp += y - 1;
    if (0 == 0) temp += 0; else temp += 0 - 1;
    if (y+1 == 0) temp += 0; else temp += (y + 1) - 1;
    return temp;
}
```

This makes further optimisations possible:
```c
int f(int y) {
    int temp = 0;
    if (y == 0) {} else temp += y - 1;
    if (y == -1) {} else temp += y;
    return temp;
}
```

```c
int f(int y) {
    int temp = 0;
    if (y == 0) {} else temp = y - 1;
    if (y == -1) {} else temp += y;
    return temp;
}
```

This can be further simplified to:
```c
int f(int y) {
    if (y == 0) return 0;
    if (y == -1) return (y-1); else return(y*y-1);
}
```
Loop unrolling:
Replace loop with repeated loop bodies:

```c
for (i=0; i<10; i++) {
    a[i]++;
    a[i]++;
    ...a[0]++;
    ...a[1]++;
    ...a[8]++;
    ...a[9]++;
    ...}
```

Why is this an optimisation?

Optimisations:
- Sometimes complicated trade-offs involved (e.g., memory bandwidth vs CPU)
- Interactions between different optimisations hard to predict
- When should which optimisation happen?
- Don’t improve inefficient algorithms!

Code Generator:
- Often relatively ad-hoc
- Interaction between different optimisations hard to predict
- Almost all optimisations are subject to trade-offs
Can be seen as abstract machine running on top of the concrete machine:

- has to lex and parse expressions as well
- cannot do global optimisations
- in general, less efficient than compiler
- useful for testing and debugging
- better portability