IMPLEMENTATION OF PROGRAMMING LANGUAGES

User
Source Language

Machine Code
Computer

→ various ways to bridge this gap:
  - Compiler
  - Interpreter
  - Hybrid approaches
Compiler translates source language into machine code
Interpreter as an abstract machine on top of the concrete machine, executes the source language instructions.
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
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 extend 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;
..."

→ comments, spaces, new lines are discarded
→ language specific
Decomposes string into sequence of tokens

"int foo() {\n int i;...

Lexer

Ident "int"  Ident "foo"  LParen  RParen  LBrace  Ident "int"  Ident "i"  Semicolon

- comments, spaces, new lines are discarded
- language specific
- How would you implement a data type `Token` in C, Java, or Haskell?
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
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)
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 stmt1 else stmt2; | return expr; | { locDec stmts } | while ( expr ) stmt
stmts ::= ε | stmt stmts
expr ::= Num | Ident | expr1 + expr2 | expr1 - expr2 | Ident = expr | Ident ( exprs )
locDec ::= Ident1 Ident2 ;
arguments ::= ε | ... 
```

What would be a suitable data type definition for a parse tree?
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
Most compilers implement some kind of optimisations
often, intermediate code is generated for this purpose
intermediate code 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:

```cpp
int pred(int x) { if (x == 0) return 0; else return x - 1; }
```

Before inlining:

```cpp
int f(int y) {
    return pred(y) + pred(0) + pred(y+1);
}
```

After inlining:

```cpp
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;
}
```

*Example taken from wikipedia.org*
This makes further optimisations possible:

```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;
}
```

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

can be further simplified to
int f(int y) {
    int temp = 0;
    if (y == 0) {} else temp = y - 1;
    if (y == -1) {} else temp += y;
    return temp;
}

can be further simplified to

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]++;
}
```

Why is this an optimisation?
Loop unrolling:

Replace loop with repeated loop bodies:

\[
\begin{align*}
\vdots \\
\vdots \\
\vdots \\
\text{for } (i=0; i <10; i++) \\ a[i]++ \\
\vdots \\
\vdots \\
\vdots \\
\end{align*}
\]

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

Attributed Parse Tree

\[ \text{Code Generator} \]

Machine Code
INTERPRETER

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