COMP 520 - Compilers

Lecture 6 (Tue Feb 1, 2022)

Structure and Operation of Compilers

• PA1 due date: extended to Wed Feb 2, 11.59 PM
  – Submission instructions on slides 2 – 5 below

• Reading assignment for Thu Feb 3
  – Skim secn 4.4 pp 109 -118 (Abstract Syntax Trees)
PA1 submission

- Java project structure for pa1
  - Package names
    - miniJava
    - miniJava.SyntacticAnalyzer
  - Main class
    - Compiler.java in package miniJava

```
package miniJava;

package miniJava.SyntacticAnalyzer;
```
Sample compiler.java

```java
package miniJava;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import miniJava.SyntacticAnalyzer.Parser;
import miniJava.SyntacticAnalyzer.Scanner;

public class Compiler {
    public static void main(String[] args) {
        InputStream inputStream = null;
        try {
            inputStream = new FileInputStream(args[0]);
        } catch (FileNotFoundException e) {
            System.out.println("Input file " + args[0] + " not found");
            System.exit(1);
        }

        ErrorReporter errorReporter = new ErrorReporter();
        Scanner scanner = new Scanner(inputStream, errorReporter);
        Parser parser = new Parser(scanner, errorReporter);

        System.out.println("Syntactic analysis ... ");
        parser.parse();
        System.out.println("Syntactic analysis complete: ");
        if (errorReporter.hasErrors()) {
            System.out.println("Invalid miniJava program");
            System.exit(4);
        } else {
            System.out.println("Valid miniJava program");
            System.exit(0);
        }
    }
}
```

return exit code 1 if unable to open input file

return these exit codes for invalid / valid miniJava programs
Project submission on server

• Submission instructions
  1. You do not have a home directory on comp520-1sp22.cs.unc.edu so run the simple readiness check pa1.pl on your own machine or a department server. You’ll receive either
    • an indication that your submission passes the top level check – Great! you’re all set!
    • an error in one of several simple miniJava programs -- correct this and rerun
  2. Copy your miniJava directory and subdirectories directly into your pa1 submission folder (once it has been created)
    • scp -pr miniJava comp520-1sp22.cs.unc.edu/home/submit/onyen/pa1

• Advice
  – try uploading something before the 11.59 pm deadline to avoid unwelcome surprises
  – The grader will run sometime after the deadline and will generate a test report in your pa1 submission directory

• Note: check Piazza for info/updates
How compilers are organized

• Today’s topics
  • Steps in the compilation process
  • Illustration of the steps for a Triangle program
  • parts of the compiler and their relation to the compiler specification
  • a bird’s eye view of implementation
# Phases of compilation

<table>
<thead>
<tr>
<th>Phase</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexical analysis (scanner)</td>
<td>character stream</td>
<td>tokens</td>
</tr>
<tr>
<td>syntactic analysis (parser)</td>
<td>tokens</td>
<td>abstract syntax tree (AST)</td>
</tr>
<tr>
<td>Contextual analysis</td>
<td>AST</td>
<td>decorated AST</td>
</tr>
<tr>
<td>(type checker / identifier resolution)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>optional Optimization</td>
<td>decorated AST</td>
<td>decorated AST</td>
</tr>
<tr>
<td>Code generation</td>
<td>decorated AST</td>
<td>machine instructions</td>
</tr>
</tbody>
</table>

Unified in PLPJ text

Front end

Back end
FE: Lexical Analysis (Triangle)

- Recognize the meaningful units in the source code
  - input: character stream
    ```
    ! this is part of a Triangle program
    while b do
    begin
      n := 0;
      b := false
    end
    ```
  - output: token stream
    ```
    while  b  do begin
      n := 0;
      b := false
    end
    ```

```sql
while  b  do begin
  n := 0;
  b := false
end
```
FE: Syntactic Analysis (Triangle)

- Use Triangle grammar to parse token stream into (concrete) syntax tree

Program ::= Single-Command eot
Command ::= Single-Command | Command ; Single-Command
Single-Command ::= while Expression do Single-Command | V-name ::= Expression
                | begin Command end | ...
Expression ::= Primary-Expression | ...
Primary-Expression ::= intLit | V-name | ...
V-name ::= ident

![Syntax Tree Diagram]

Program
**FE: Construct Abstract Syntax Tree**

- While parsing concrete syntax tree, construct the abstract syntax tree

---

**AST “grammar” for mini-Triangle**
FE: Construct Abstract Syntax Tree (Triangle)

Concrete Syntax Tree

Abstract Syntax Tree
FE: Contextual analysis

- Traverse AST
  - determine the declaration associated with each identifier referenced
  - determine the type of all expressions
  - record in AST

```
let
  var n: Integer;
  var c: Char
in
  begin
    c := '&';
    n := n + 1
  end
```

Decorated AST for sample program
BE: Optimization

- Restructure AST so that it corresponds to an equivalent but more efficient program
  - simple example: constant folding
    \[ x := y \times 0 \quad \Rightarrow \quad x := 0 \]
  - introduce temporaries to hold previously computed values

\[
\begin{align*}
  f[i] &:= f[i] + (m[i] \times m[j]) / \text{pow}(x[i] - x[j], 2); \\
f[j] &:= f[j] - (m[i] \times m[j]) / \text{pow}(x[i] - x[j], 2);
\end{align*}
\]
BE: Code Generation

- **Produce machine code**
  - for abstract machine or physical machine

- **Issues**
  - location of program variables (stack, heap)
  - instruction selection and register allocation
  - linkage conventions (ABI: Application Binary Interface)

Triangle

```
let
  var n: Integer
  var c: Char
in
  begin
    c := ' &';
    n := n + 1
  end
```

TAM instructions

```
PUSH 2
LOADL 38
STORE 1[ SB]
LOAD 0[ SB]
LOADL 1
CALL add
STORE 0[ SB]
POP 2
HALT
```
Implementation of multiple phases

• Single-pass compiler
  – economical in time and space
  – limited ability to implement language features and optimizations

• Multi-pass compiler
  – conceptually simpler and more versatile
  – requires more space and time

• Wirth’s design principle
  – design programming languages so that their compilers are simple
    • Pascal
    • C + “.h” header files

• Programmers design principle
  – but not too simple!
Traditional Two-Pass Compiler

- **Front end** creates AST
  - time complexity $O(n)$ or $O(n \log n)$ where $n$ is size of the program in characters

- **Back end** translates AST to target machine
  - $O(n)$ or $O(n \log n)$ time complexity for simple code generation
  - but most back end optimization problems are NP-hard
UNCOL: the universal AST?

• Suppose we have
  – $n$ programming languages
  – $m$ target machines
  – do we really need to construct $n \times m$ compilers?

• A *universal* AST would allow us to
  – construct $n$ front-ends
  – construct $m$ back ends
  – $n + m$ components: much less work!

• The *Universal Computer Oriented Language*: an elusive goal since 1960
  – variation (evolution) in programming languages
  – variation (evolution) in hardware architecture
  – but hope springs eternal: JVM? .NET?
Back End (Instruction Selection)

- Produce compact, fast code
- Use available addressing modes
- Pattern matching problem
  - \textit{ad hoc} techniques
  - tree pattern matching
  - dynamic programming

IR $\xrightarrow{\text{Instruction selection}}$ Register allocation $\xrightarrow{\text{machine code}}$ errors
Back End (Register Allocation)

- performance advantage if a register is used instead of memory
- but we have a limited number of registers
  - which values to keep in a register?
- optimal allocation difficult
  - NP-complete for \( k \geq 1 \) registers
Compilers and Instruction Set Architecture

- Compiler benefits from simple instruction set
  - Difficult to deal with complex operations in instruction set
    - example: VAX instruction INDEX(base-addr, i, low, high)
      - if (low \leq i \leq high) return base-addr + 4 * i
      - 2 comparisons, 1 multiply, 1 add

- Typical program
  - Only one test necessary
    - loop bounds guarantee all values of i are valid indices
  - No multiplications necessary
    - AddressOf(a[i+1]) == AddressOf(a[i])+4
  - better to avoid INDEX operation

- Current processor architectures
  - “orthogonal” RISC instruction set
    - easy for compiler to generate and optimize
    - easy to implement in hardware with superior performance

```c
int a[10]; int s = 0;
for (i = 0; i < 10; i++)
    s += a[i];
end
```