COMP 520 - Compilers

Lecture 6 (Thu January 28, 2016)

Compiler Design and Implementation

• Please pick up from rear of class
  – Graded WA2

• No new reading assignment for next Tuesday

• PA1 due Monday Feb 1 by midnight.
Overview

• Today’s topics
  – Finish scanner discussion
    • Examine Triangle scanner
    • Automatic scanner generation
  – How compilers are organized
    • 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><strong>lexical analysis</strong></td>
<td>character stream</td>
<td>tokens</td>
</tr>
<tr>
<td><em>(scanner)</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>syntactic analysis</strong></td>
<td>tokens</td>
<td>abstract syntax tree (AST)</td>
</tr>
<tr>
<td><em>(parser)</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Contextual analysis</strong></td>
<td>AST</td>
<td>decorated AST</td>
</tr>
<tr>
<td><em>(type checker / identifier resolution)</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>optional</strong></td>
<td>decorated AST</td>
<td>decorated AST</td>
</tr>
<tr>
<td><strong>Optimization</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Code generation</strong></td>
<td>decorated AST</td>
<td>machine instructions</td>
</tr>
</tbody>
</table>

Unified in PLPJ text

**Front end:**

**Back end:**
FE: Lexical Analysis

- 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
    ```
FE: Syntactic Analysis

- Use 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

Program

eot
FE: Construct Abstract Syntax Tree

- While parsing concrete syntax tree, construct the more useful abstract syntax tree.

```
Program ::= Command

Command ::= V-name ::= Expression | Identifier ( Expression ) | Command ; Command | if Expression then Command else Command | while Expression do Command | let Declaration in Command

Expression ::= Integer-Literal | V-name | Operator Expression | Expression Operator Expression

V-name ::= Identifier

Declaration ::= const Identifier ::= Expression | var Identifier : Type-denoter | Declaration ; Declaration

Type-denoter ::= Identifier
```

AST “grammar” for a part of mini-Triangle
FE: Construct Abstract Syntax Tree (2)

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

 decorated AST for sample program
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)

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

```
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
  – we 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: .net ? JVM ?