APT Session 7: Compilers

Laurence
Tratt

KING'S College LONDON

Software Development Team
2017-12-01

What to expect from this session

1. Building a compiler.

Prerequisites

1. Have the programming language of your choice (e.g. Java, Python) installed and running on your computer.
2. For Java: clone and compile Gabriela Moldovan's fork of Sam White's parser here.
3. For Python: download my simple parser.
4. Use Sam White's stack visualizer.

Compilers (1)

- Previously we wrote a simple stack machine.

Compilers (1)

- Previously we wrote a simple stack machine.
- A compiler can take in ‘human readable’ programs and convert them to the stack machine format.

Compilers (1)

- Previously we wrote a simple stack machine.
- A compiler can take in ‘human readable’ programs and convert them to the stack machine format.
- Compilers for dynamically typed languages are simple. [Statically typed languages’ compilers, especially if they heavily optimise code, are superficially more complex, though they’re still conceptually simple.]

Compilers (1)

- Previously we wrote a simple stack machine.
- A compiler can take in 'human readable' programs and convert them to the stack machine format.
- Compilers for dynamically typed languages are simple. [Statically typed languages' compilers, especially if they heavily optimise code, are superficially more complex, though they're still conceptually simple.]
- Our starting grammar:

```
Stmt   ::= Assign | While | Print
Assign ::= ID = Expr ;
Expr   ::= INT + Expr | INT - Expr | INT < Expr
           | INT > Expr | INT == Expr | INT
While  ::= WHILE Expr { stmt* }
Print  ::= PRINT Expr ;
```

- Python and Java tree-creating parsers are provided for this grammar.

Compilers (2)

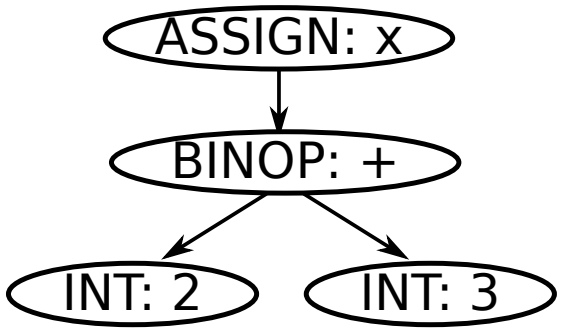
- The transformation our compiler needs is fairly simple. For example `print 2 + 3;` should be transformed to:

```
INT 2
INT 3
ADD
PRINT
EXIT
```

- The trick is to break this transformation into phases. The first is parsing (which we've done for you). The second is code generation (which we'll do today). The (optional) third is optimisation (which we won't cover).

Parse trees

- A parser takes a grammar and input text and produces a tree.
- For example the parse tree for `x = 2 + 3;` is:

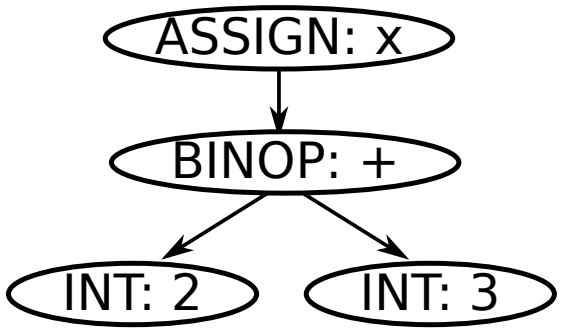


  which we will write as `assign("x", binop("+", int(2), int(3)).`

- The program `print 2;` is transformed to the parse tree `print(int(2))` etc.

Parse trees

- A parser takes a grammar and input text and produces a tree.
- For example the parse tree for `x = 2 + 3;` is:



  which we will write as `assign("x", binop("+", int(2), int(3)).`

- The program `print 2;` is transformed to the parse tree `print(int(2))` etc.

Exercises:

1 Write the program `print 2;` into a file `ex1.hll` and parse it.

Traversing over a parse tree

- A compiler is a translator of parse tree nodes to code.

Traversing over a parse tree

- A compiler is a translator of parse tree nodes to code.
- To make life easier, parse tree edges are labelled. So a `print` node has an `exp`, an `int` node has a `val` and so on.

Traversing over a parse tree

- A compiler is a translator of parse tree nodes to code.
- To make life easier, parse tree edges are labelled. So a `print` node has an `exp`, an `int` node has a `val` and so on.
- We iterate over the tree, gradually converting it to stack-based instructions.
- For a print node: process the `exp` then add a `PRINT` instruction. For an int: simply create an `INT` instruction.

Traversing over a parse tree

- A compiler is a translator of parse tree nodes to code.
- To make life easier, parse tree edges are labelled. So a `print` node has an `exp`, an `int` node has a `val` and so on.
- We iterate over the tree, gradually converting it to stack-based instructions.
- For a print node: process the `exp` then add a `PRINT` instruction. For an int: simply create an `INT` instruction.

Exercises:

1 Write a simple compiler which reads a file in, parses it, converts `ex1.hll` to a stack-based format and prints the output to `stdout`. Initially you only need to handle parse trees of the form `print(int(`*n*`))` (where *n* is an integer). Run the output in the stack visualiser.

More sophisticated traversal

- Compilers mostly do a *preorder* traversal: process the node; process the Left Hand Side (LHS) of the node (all the way); process the Right Hand Side (RHS) of the node (all the way).
- For a `BINOP` (e.g. add/sub/etc.) node: process the `lhs` then the `rhs` then emit the appropriate binop instruction.

More sophisticated traversal

- Compilers mostly do a *preorder* traversal: process the node; process the Left Hand Side (LHS) of the node (all the way); process the Right Hand Side (RHS) of the node (all the way).
- For a `BINOP` (e.g. add/sub/etc.) node: process the `lhs` then the `rhs` then emit the appropriate binop instruction.
- For each node type *X*, we need a function `t_`*X* which to process it. We also need a function `preorder` which takes in an arbitrary node and dispatches to the appropriate `t_` function.

More sophisticated traversal

- Compilers mostly do a *preorder* traversal: process the node; process the Left Hand Side (LHS) of the node (all the way); process the Right Hand Side (RHS) of the node (all the way).
- For a `BINOP` (e.g. add/sub/etc.) node: process the `lhs` then the `rhs` then emit the appropriate binop instruction.
- For each node type *X*, we need a function `t_`*X* which to process it. We also need a function `preorder` which takes in an arbitrary node and dispatches to the appropriate `t_` function.

Exercises:

1. Adjust your previous compiler to be a class with traversal functions `t_` and a general `preorder` function.
2. Write the program `print 2 + 3` into a file `ex2.hll`.
3. Add a binop traversal function (which need only cope with '+').
4. Run the output in the stack visualiser.

Variables

- Working only with the stack is frustrating. We treat variables as named parts of the *heap*.
- The stack machine has two variable instructions:

```
VAR_SET x
VAR_LOOKUP y
```

`VAR_SET` sets the variable `x` to the (popped) top of the stack.
`VAR_LOOKUP` looks up a variable `y` and pushes it onto the stack.

Variables

- Working only with the stack is frustrating. We treat variables as named parts of the *heap*.
- The stack machine has two variable instructions:

```
VAR_SET x
VAR_LOOKUP y
```

`VAR_SET` sets the variable `x` to the (popped) top of the stack.
`VAR_LOOKUP` looks up a variable `y` and pushes it onto the stack.

Exercises:

1. Write the program `x=2; y=2+3; print x+y;` into a file `ex3.hll`.
2. Add variable traversal functions.
3. Run the output in the stack visualiser.

Conditionals

- The stack machine defines `LESS_THAN`, `GREATER_THAN`, and `EQUALS` which do:

```
rhs = stack.pop()
lhs = stack.pop()
stack.push(lhs op rhs)
```

  where `op` is <, >, or ==. 0 is pushed for false, 1 for true.

Conditionals

- The stack machine defines `LESS_THAN`, `GREATER_THAN`, and `EQUALS` which do:

```
rhs = stack.pop()
lhs = stack.pop()
stack.push(lhs op rhs)
```

  where `op` is <, >, or ==. 0 is pushed for false, 1 for true.

Exercises:

1. Write the program `print 2 < 3;` into a file `ex4.hll`.
2. Add conditional traversal functions.
3. Run the output in the stack visualiser.

Loops

- A `while` loop has a condition and a body. While the condition doesn't evaluate to 0, the body is executed.

Loops

- A `while` loop has a condition and a body. While the condition doesn't evaluate to 0, the body is executed.

Exercises:

1. Write the program

```
i = 0;
while i < 10 {
    print i;
    i = i + 1;
}
```

   into a file `ex5.hll`.

2. Add a while loop traversal function. You will need labels and jumps.

3. Run the output in the stack visualiser.

Post-session exercises

Try these (no particular order):

- Extend the parser to handle functions and update the compiler accordingly.
- Read how difficult is it to write a compiler? in light of your experiences.