Interpreter

Motivation

- What can we do with the parse tree?
- How about running the program we've parsed?
- This is where our compiler code is no longer independent of the grammar we're using

Grammar

- Recall: The grammar:

IF → \bif\b
ELSE → \belse\b
WHILE → \bwhile\b
SEMI → ;
NUM → \d+
ID → \w+
EQ → =
LP → [(]
RP → [)]
ADDOP → [-+]
MULOP → [*/]
LBR → [{]
RBR → [}]

S → stmt SEMI S | λ
a-o-f → ID a-o-f'
a-o-f' → EQ e | LP a-o-f''
a-o-f'' → RP | e RP
cond → IF LP e RP LBR S RBR cond'
cond' → λ | ELSE LBR S RBR
e → t e'
e' → ADDOP t e' | λ
f → ID | NUM | LP e RP
loop → WHILE e LBR S RBR
stmt → a-o-f | cond | loop
t → f t'
t' → MULOP f t' | λ

Table

	$	ADDOP	ELSE	EQ	ID	IF	LBR	LP	MULOP	NUM	RBR	RP	SEMI	WHILE
S	λ	●	●	●	stmt SEMI S	stmt SEMI S	●	●	●	●	λ	●	●	stmt SEMI S
a-o-f	●	●	●	●	ID a-o-f'	●	●	●	●	●	●	●	●	●
a-o-f'	●	●	●	EQ e	●	●	●	LP a-o-f''	●	●	●	●	●	●
a-o-f''	●	●	●	●	e RP	●	●	e RP	●	e RP	●	RP	●	●
cond	●	●	●	●	●	IF LP e RP LBR S RBR cond'	●	●	●	●	●	●	●	●
cond'	●	●	ELSE LBR S RBR	●	●	●	●	●	●	●	●	●	λ	●
e	●	●	●	●	t e'	●	●	t e'	●	t e'	●	●	●	●
e'	●	ADDOP t e'	●	●	●	●	λ	●	●	●	●	λ	λ	●
f	●	●	●	●	ID	●	●	LP e RP	●	NUM	●	●	●	●
loop	●	●	●	●	●	●	●	●	●	●	●	●	●	WHILE e LBR S RBR
stmt	●	●	●	●	a-o-f	cond	●	●	●	●	●	●	●	loop
t	●	●	●	●	f t'	●	●	f t'	●	f t'	●	●	●	●
t'	●	λ	●	●	●	●	λ	●	MULOP f t'	●	●	λ	λ	●

Interpreter

- We'll create a very simple interpreter that just walks over the tree and executes statements
- Creating interpreter is straightforward, but tedious: We need to write case statements for every grammar production

Code

- Suppose our interpreter is structured like so:

```
def interpret(n):
    #n = TreeNode: Has fields
    #   sym (string)
    #   token (Token),
    #   children (list of TreeNode)
    t=n.sym
    ch=n.children
    ...do something...
```

Code

- Begin with the easy cases:

```
if t=="S":
    if len(ch) > 0:     # S -> stmt SEMI S
        interpret(ch[0])
        interpret(ch[2])
    else:
        pass            # S -> lambda
elif t=="stmt":         # stmt -> aof | cond
    interpret(ch[0])
```

Code

- Now we have these:
  a-o-f $\rightarrow$ ID a-o-f'
  a-o-f' $\rightarrow$ EQ e | LP a-o-f''
  a-o-f'' $\rightarrow$ e RP | RP
- If we're processing a-o-f node: We don't know what to do quite yet
- We could have a-o-f examine its child
  - But that's a bit messy: We're conflating different nodes' processing together
- We could have a-o-f' look at its sibling
  - But that's also messy
  - And a-o-f'' would need to consider its uncle: More convoluted

Attributes

- We introduce the concept of *attributes*
- Two types: *inherited* and *synthesized*
  - Inherited = passed down the tree
  - Synthesized = passed up the tree

Define

- We need to define the attributes of each production
- None of the productions we've seen so far have any attributes
- We'll introduce some now...

Attributes

- Productions:
  - a-o-f $\rightarrow$ ID a-o-f'
    - Sends data to children
  - a-o-f' $\rightarrow$ EQ e
    - Inherited attribute: The ID from its parent (a-o-f)
  - a-o-f' $\rightarrow$ LP a-o-f"
    - Inherited attribute: The ID from its parent (a-o-f). Sends to its child
  - a-o-f" $\rightarrow$ RP
    - Inherited attribute: The ID from its parent
  - a-o-f" $\rightarrow$ e RP
    - Inherited attribute: The ID from its parent

Code

- To implement this, we need to change signature of interpret()
  def interpret(n , inherited=None)
  - inherited will contain any inherited attributes
  - interpret() will return any synthesized attributes

Code

- Now we can write our handler:

```
elif t == "a-o-f":
    interpret( ch[1], ch[0].token.lexeme )
```

a-o-f'

- a-o-f' $\rightarrow$ EQ e
- This involves expression (e)
- We haven't looked at expression (e) yet, but assume it has a synthesized attribute: The value of the expression

```
elif t == "a-o-f'":
    if ch[0].sym == "EQ":        #EQ e
        val = interpret( ch[1] )
        ...what now?...
```

Assignment

- Interpreter will maintain a symbol table: A dictionary
- Key = variable name; value = its value
  - Our language only supports numerical values
  - If we had several types, the symbol table would also keep track of the variable's type

```
elif t == "a-o-f'":
    if ch[0].sym == "EQ":   #EQ e
        val = interpret( ch[1] )
        symtable[ inherited ] = val
    else:   # LP a-o-f''
        ...what now?...
```

a-o-f’

- Our grammar doesn’t allow defining of custom functions
- So we have to specify some
- We’ll give two: write() and halt()
- Semantics:
  - write() takes one argument and outputs this value to the console
  - halt() takes no arguments and terminates the program

a-o-f'

```
elif t == "a-o-f'":
    ...
    else:   # LP a-o-f''
        interpret( ch[1], inherited )
```

a-o-f”

- We can now finish off the function processing

```
elif t == "a-o-f''":
    if len(ch) == 1:    # e RP
        val = interpret(ch[0])
        if inherited == "write":
            print(val)
        else:
            error!
    else:           #RP
        if inherited == "halt":
            sys.exit(0)
        else:
            error!
```

Numeric

- We can now begin working on the expression (e) / term (t) / factor (f) chain
- All of these will have synthesized attributes that represent the value of the thing evaluated
- Begin at the bottom of the chain: f

f

- f $\rightarrow$ ID | NUM | LP e RP
- If it's a number: Easy: Just return the value
- If it's LP e RP: Also easy: Evaluate the expression and return its synthesized attribute
- If it's ID:
  - Look it up in symbol table
  - If not found: Error
  - Else, return its value
  - Since only one variable type, no need to do type checking

Code

```
elif t == "f":
    if ch[0].sym == "NUM":
        return int(ch[0].token.lexeme)
    elif ch[0].sym == "ID":
        varname = ch[0].token.lexeme
        if varname not in symtable: error
        else: return symtable[varname]
    else:
        return interpret(ch[1])
```

t

- We now have a tandem:
  - t $\rightarrow$ f t'
  - t' $\rightarrow$ MULOP f t' | λ
- This is a little trickier. We need to make use of both inherited and synthesized attributes

```
elif t == "t":
    v = interpret(ch[0])        #the synthesized attr...
    return interpret(ch[1],v)   #...becomes an inherited attr
```

t'

- t' $\rightarrow$ MULOP f t' | λ

```
elif t == "t'":
    if len(ch) > 0:
        op = ch[0].token.lexeme
        v2 = interpret(ch[1])
        if op == "*":
            v = inherited * v2
        elif op == "/":
            v = inherited / v2
        else: ICE       #internal compiler error
        return interpret( ch[2], v )     #data flows up and down
                                         #the tree...
    else:       #lambda
        return inherited    #data bounces back up
```

e and e'

- Same idea as t and t' just with different op's

cond & cond'

- cond $\rightarrow$ IF LP e RP LBR S RBR cond'
- cond' $\rightarrow \lambda$ | ELSE LBR S RBR
- By now, you probably already can guess how to proceed...

loop

- loop $\rightarrow$ WHILE e LBR S RBR
- This is easy, too!

Assignment

- Implement the complete interpreter for the language

Sources

- Aho, Lam, Sethi, Ullman. Compilers: Principles, Techniques, & Tools (2nd ed).
- K. Louden. Compiler Construction: Principles and Practice.

Created using $\LaTeX$.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org