CS 406: Crafting a Simple Compiler

Stefan D. Bruda

Winter 2016

- ac = a simple programming language to illustrate the compilation phases
  - Types: `integer` and `float`
  - Keywords: `f` (declares a float variable), `i` (declares an integer variable), and `p` (prints the value of a variable)
  - Variables: 23 possible, one-letter variable names (excluding the reserved keywords)
  - Type conversions: automatic from `integer` to `float`, forbidden the other way around
  - Operations: addition, subtraction, assignment
- Compilation: translation of an `ac` program into a `dc` program
  - `dc` = stack-based calculator using the reverse Polish notation

AC DEFINITION: TOKENS

- Tokens are the lexical units of a program
  - A token makes sense to the program as a whole but its substrings do not
- Token definition is accomplished using regular expressions

Token	Regular expression
floatdcl	$\mathtt{f}$
intdcl	$\mathtt{i}$
print	$\mathtt{p}$
id	$[\mathtt{a}-\mathtt{e}] \mid [\mathtt{g}-\mathtt{h}] \mid [\mathtt{j}-\mathtt{o}] \mid [\mathtt{q}-\mathtt{z}]$
assign	$=$
plus	$+$
minus	$-$
inum	$[0-9][0-9]^*$ or $[0-9]^+$
fnum	$[0-9]^+.[0-9]^+$
blank	$\sqcup^+$

- The syntax of a programming language is specified using a context-free grammar
- Alternate (textbook) notations:
  - Capitalized nonterminals rather than $\langle \dots \rangle$
  - λ for the empty string rather than ε

$$
\begin{array}{rcll}
\langle \text{prog} \rangle & \to & \langle \text{dcls} \rangle\ \langle \text{stmts} \rangle\ \$ & (1) \\
\langle \text{dcls} \rangle & \to & \langle \text{dcl} \rangle\ \langle \text{dcls} \rangle & (2) \\
 & | & \varepsilon & (3) \\
\langle \text{dcl} \rangle & \to & \text{floatdcl id} & (4) \\
 & \to & \text{intdcl id} & (5) \\
\langle \text{stmts} \rangle & \to & \langle \text{stmt} \rangle\ \langle \text{stmts} \rangle & (6) \\
 & | & \varepsilon & (7) \\
\langle \text{stmt} \rangle & \to & \text{id assign}\ \langle \text{val} \rangle\ \langle \text{expr} \rangle & (8) \\
 & | & \text{print id} & (9) \\
\langle \text{expr} \rangle & \to & \text{plus}\ \langle \text{var} \rangle\ \langle \text{expr} \rangle & (10) \\
 & | & \text{minus}\ \langle \text{var} \rangle\ \langle \text{expr} \rangle & (11) \\
 & | & \varepsilon & (12) \\
\langle \text{val} \rangle & \to & \text{id} & (13) \\
 & | & \text{inum} & (14) \\
 & | & \text{fnum} & (15)
\end{array}
$$

```
function SCANNER () returns Token:
    while s.PEEK() = blank do s.ADVANCE()
    if s.EOF() then ans.type ← $
    else
        if s.PEEK() ∈ {0, 1, ..., 9} then
            ans ← SCANDIGITS()
        else
            ch ← s.ADVANCE()
            switch ch do
                case {a, b, ..., z} \ {i, f, p}:
                    ans.type ← id
                    ans.val ← ch
                case f: ans.type ← floatdcl
                case i: ans.type ← intdcl
                case p: ans.type ← print
                case =: ans.type ← assign
                case +: ans.type ← plus
                case −: ans.type ← minus
                otherwise LEXICALERROR()
    return ans
```

```
function SCANDIGITS () returns Token:
    tok.val ← " "
    while s.PEEK() ∈ {0, 1, ..., 9} do
        tok.val ← tok.val + s.ADVANCE()
    if s.PEEK() ≠ . then tok.type ← inum
    else
        tok.type ← fnum
        tok.val ← tok.val + s.ADVANCE()
        while s.PEEK() ∈ {0, 1, ..., 9} do
            tok.val ← tok.val + s.ADVANCE()
    return tok
```

s = stream of characters (input, global)
PEEK() returns the next entity in the input
ADVANCE() returns and erases the next entity in the input

PARSING (SYNTACTIC ANALYSIS)

```
function MATCH(ts: TokenStream, m: Type):
    if ts.ADVANCE() ≠ m then PARSEERROR()

function STMT ():
    if ts.PEEK() = id then
        MATCH(ts,id)
        MATCH(ts,assign)
        VAL()
        EXPR()
    else if ts.PEEK() = print then
        MATCH(ts,print)
        MATCH(ts,id)
    else PARSEERROR()

function STMTS():
    if ts.PEEK() = id or ts.PEEK() = print then
        STMT()
        STMTS()
    else if ts.PEEK() = $ then
        /* Do nothing (ε rule) */
    else PARSEERROR()
```

ts = stream of tokens (input, global)

```
function MATCH(ts: TokenStream, m: Type):
    if ts.ADVANCE() ≠ m then PARSEERROR()

function STMT ():
    if ts.PEEK() = id then
        MATCH(ts,id)
        MATCH(ts,assign)
        VAL()
        EXPR()
    else if ts.PEEK() = print then
        MATCH(ts,print)
        MATCH(ts,id)
    else PARSEERROR()

function STMTS():
    if ts.PEEK() = id or ts.PEEK() = print then
        STMT()
        STMTS()
    else if ts.PEEK() = $ then
        /* Do nothing (ε rule) */
    else PARSEERROR()
```

ts = stream of tokens (input, global)

- Example of recursive descent parsing

```
function MATCH(ts: TokenStream, m: Type):
    if ts.ADVANCE() ≠ m then PARSEERROR()

function STMT ():
    if ts.PEEK() = id then
        MATCH(ts,id)
        MATCH(ts,assign)
        VAL()
        EXPR()
    else if ts.PEEK() = print then
        MATCH(ts,print)
        MATCH(ts,id)
    else PARSEERROR()

function STMTS():
    if ts.PEEK() = id or ts.PEEK() = print then
        STMT()
        STMTS()
    else if ts.PEEK() = $ then
        /* Do nothing (ε rule) */
    else PARSEERROR()
```

ts = stream of tokens (input, global)

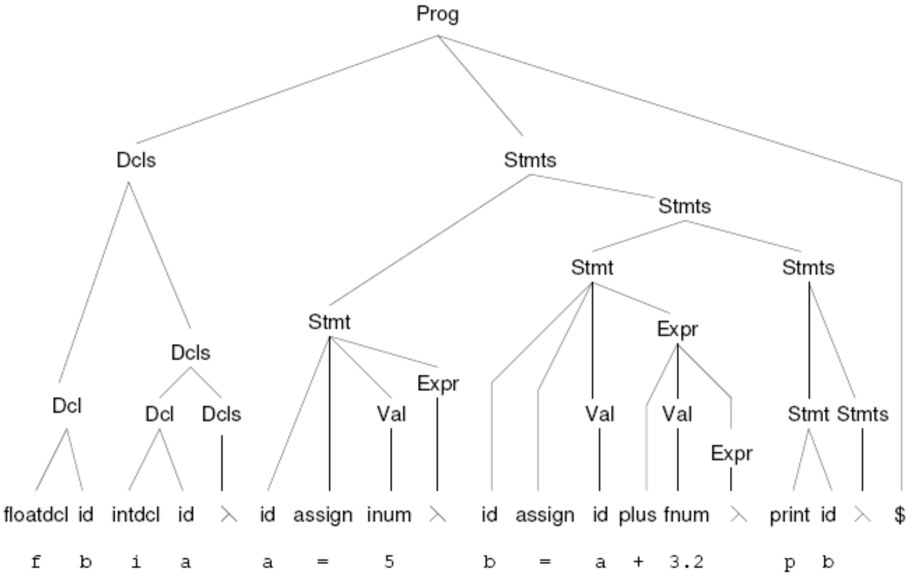
- Example of recursive descent parsing
  - Verifies that the program is syntactically correct
- Will in effect construct the parse tree (through recursion)
  - Code easily modified to produce the actual tree

```
function MATCH(ts: TokenStream, m: Type):
    if ts.ADVANCE() ≠ m then PARSEERROR()

function STMT ():
    if ts.PEEK() = id then
        MATCH(ts,id)
        MATCH(ts,assign)
        VAL()
        EXPR()
    else if ts.PEEK() = print then
        MATCH(ts,print)
        MATCH(ts,id)
    else PARSEERROR()

function STMTS():
    if ts.PEEK() = id or ts.PEEK() = print then
        STMT()
        STMTS()
    else if ts.PEEK() = $ then
        /* Do nothing (ε rule) */
    else PARSEERROR()
```

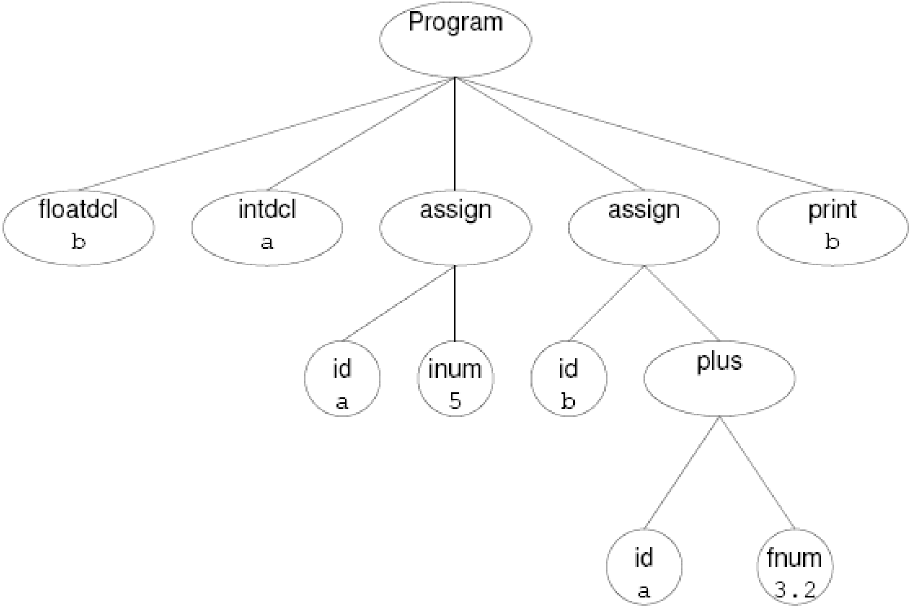
ts = stream of tokens (input, global)

- Example of recursive descent parsing
  - Verifies that the program is syntactically correct
- Will in effect construct the parse tree (through recursion)
  - Code easily modified to produce the actual tree
- In practice however an abstract syntax tree or AST is constructed instead
  - Simplified version of the parse tree, that stores the structure of the program but not the details (such as the presence of keywords or the use of ε/λ rules)
  - The AST is the end product of parsing and is passed along to the next phase

Example of ac Program and Its Parse Tree



Example of ac Abstract Syntax Tree



SEMANTIC ANALYSIS

- The semantic analysis examines the program properties that are not context free (and so cannot be verified by the parser).
  - Declarations and scope (using a symbol table)
  - Type checking
  - Other semantic rules
- Semantic analysis is generally accomplished using walks on the AST

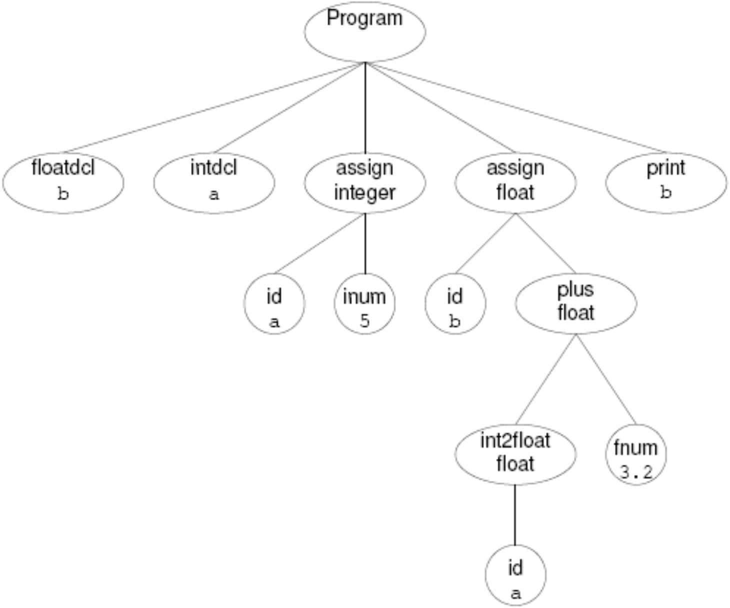
- The semantic analysis examines the program properties that are not context free (and so cannot be verified by the parser).
  - Declarations and scope (using a symbol table)
  - Type checking
  - Other semantic rules
- Semantic analysis is generally accomplished using walks on the AST
- Declarations and scope: ensure that all identifiers are declared before use
  - Construction of the symbol table, accomplished by traversing the AST and processing all the nodes of type SymDecl:

```
function VISIT(n: SYMDECL):
    if n.type = floatdcl then ENTERSYMBOL(n.id, float)
    else ENTERSYMBOL(n.id, integer)

function ENTERSYMBOL(name, type):
    if SymbolTable[name] = null then SymbolTable[name] ← type
    else ERROR("Duplicate declaration")

function LOOKUPSYMBOL(name) returns Type:
    return SymbolTable[name]
```

SEMANTIC ANALYSIS (CONT'D)

```
function VISIT(n: Computing):
  n.type ← CONSISTENT(n.child1, n.child2)
function VISIT(n: Asigning):
  n.type ← CONVERT(n.child2, n.child1.type)
function VISIT(n: SymRef):
  n.type ← LOOKUPSYMBOL(n.id)
function CONSISTENT(c1, c2) returns Type:
  m ← GENERALIZE(c1.type, c2,type)
  CONVERT(c1,m)
  CONVERT(c2,m)
  return m
function GENERALIZE(t1, t2) returns Type:
  if t1 = float or t2 = float then
    and ← float
  else  and ← integer
  return ans
function CONVERT(n, t):
  if n.type = float and t = integer then
    ERROR("Illegal type conversion")
  else if n.type = integer and t = float then
    convert n to float
```

- Declarations have been processed
- Now the other nodes are visited
  - Identifiers appearing in the code are checked against the symbol table (declarations)
  - Arithmetic and assignment nodes are checked for type consistency
  - Implicit type conversion is performed from integer to float (but not the other way around)
- Semantic analysis can be accomplished with a single traversal of the AST or with multiple traversals
  - Language dependent


Program
floatdcl
b
intdcl
a
assign
integer
assign
float
print
b
id
a
inum
5
id
b
plus
float
int2float
float
fnum
3.2
id
a

CODE GENERATION

```
function CODEGEN(n: Assigning):
    CODEGEN(n.child2)
    EMIT("s")
    EMIT(n.child1.id)
    EMIT("0 k")
function CODEGEN(n: Computing):
    CODEGEN(n.child1)
    CODEGEN(n.child2)
    EMIT(n.operation)
function CODEGEN(n: SymRef):
    EMIT("l")
    EMIT(n.id)
function CODEGEN(n: Printing):
    EMIT("l")
    EMIT(n.id) EMIT("p")
    EMIT("si")
function CODEGEN(n: Converting):
    CODEGEN(n.child1)
    EMIT("5 k")
function CODEGEN(n: Const):
    EMIT(n.val)
```

Source	Code
a = 5	5 sa 0 k
b = a + 3.2	la 5 k 3.2 + sb 0 k
p b	lb p si

- Ad-hoc generation
- Recursive process
- Principles will hold for complex languages