Abstract Syntax Trees

- Our Exp1bytecode language was so straightforward that the best IR was an abstract representation of the instructions.
- In more complex languages, especially higher level languages it usually not possible to design such a simple IR.
- Instead we use Abstract Syntax Trees (ASTs).
Abstract Syntax Trees

- One way to think about ASTs is as parse trees with all the derivation information deleted.
Abstract Syntax Trees

- Because every valid program has a parse tree, it is always possible to construct an AST for every valid input program.
- In this way ASTs are the IR of choice because it doesn’t matter how complex the input language, there will always be an AST representation.
- Besides being derived from the parse tree, AST design typically follows three rules of thumb:
  - *Dense*: no unnecessary nodes
  - *Convenient*: easy to understand, easy to process
  - *Meaningful*: emphasize the operators, operands, and the relationship between them; emphasize the computations
A convenient way to represent AST nodes is with the following structure,

- \((\text{TYPE}, \text{child1, child2, ...})\)

A tree node is a tuple where the first component represents the type or name of the node followed by zero or more components each representing a child of the current node.

Consider the abstract syntax tree for \(+ x - y \ x,\)

```
In [2]: ast = ('+', 'x', ('-', 'y', 'z'))
In [3]: from grammar_stuff import dump_AST
dump_AST(ast)

(+ x
 |(- y z))
```
The Cuppa1 Language

- Our next language is a simple high-level language that supports structured programming with ‘if’ and ‘while’ statements.
- However, it has no scoping and no explicit variable declarations.
The Cuppa1 Language

Infix Expressions!

Infix Expressions!

Macmillan Computer Library

Precedence & Associativity Table:

```
program : stmt_list

stmt_list : stmt stmt_list  | empty

stmt : ID '=' exp opt_semi   | GET ID opt_semi
     | PUT exp opt_semi
     | WHILE '(' exp ')' stmt
     | IF '(' exp ')' stmt opt_else
     | '{' stmt_list '}'

opt_else : ELSE stmt
         | empty

opt_semi : ';'  | empty

exp : exp PLUS exp  | exp MINUS exp
    | exp TIMES exp
    | exp DIVIDE exp
    | exp EQ exp
    | exp LE exp
    | INTEGER
    | ID
    | '{' exp '}'
    | MINUS exp %prec UMINUS
    | NOT exp
```

```
// list of integers
get x;
while (1 <= x)
{
    put x;
    x = x - 1;
}
```
The Cuppa1 Language

```python
# grammar for Cuppa1
from ply import yacc
from cuppa1_lex import tokens, lexer

# set precedence and associativity
# NOTE: all arithmetic operator need to have tokens
# so that we can put them into the precedence table
precedence = ( ('left', 'EQ', 'LE'), ('left', 'PLUS', 'MINUS'), ('left', 'TIMES', 'DIVIDE'), ('right', 'UMINUS', 'NOT') )

def p_grammar(_):
    '''
    program : stmt_list
    stmt_list : stmt stmt_list
               | empty

    stmt : ID '=' exp opt_semi
         | GET ID opt_semi
         | PUT exp opt_semi
         | WHILE '(' exp ')' stmt
         | IF '(' exp ')' stmt opt_else
         | '!' stmt_list

    opt_else : ELSE stmt
             | empty

    opt_semi : ';'
             | empty
    '''

def p_empty(p):
    'empty :'

def p_error(t):
    print("Syntax error at '%s'") % t.value

### build the parser
parser = yacc.yacc()

The Parser Specification
```
# Lexer for Cuppa1

```python
from ply import lex

reserved = {
    'get': 'GET',
    'put': 'PUT',
    'if': 'IF',
    'else': 'ELSE',
    'while': 'WHILE',
    'not': 'NOT'
}

literals = [';', '=', '(', ')', '{', '}']

tokens = ['PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EQ', 'LE', 'INTEGER', 'ID', ] + list(reserved.values())

# build the lexer
lexer = lex.lex(debug=0)
```

The Lexer Specification

```python
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID')  # Check for reserved words
    return t
def t_INTEGER(t):
    r'[0-9]+'
    return t
def t_COMMENT(t):
    r'//.*'
    pass
def t_NEWLINE(t):
    r'\n'
    pass
def t_error(t):
    print("Illegal character \%s" % t.value[0])
    t.lexer.skip(1)
```
Testing our Parser

```
In [4]: from cuppal_gram import parser
    
from cuppal_lex import lexer

Generating LALR tables
WARNING: 1 shift/reduce conflict
```

```
In [6]: fact = 
   ...  
get x;
y = 1;
while (l <= x)
{
    y = y * x;
x = x - 1;
}
puy y;
... 

terse.parse(fact, lexer=lexer)
```

Notice the shift/reduce conflict!

The error is due to the if-then-else statement with the optional else.

The default action for shift/reduce conflicts is to always `shift`.

That is exactly right for us!

```
In [7]: loop = "while (l) {}"

parser.parse(loop, lexer=lexer)
```
The Cuppa1 Frontend

- A frontend is a parser that
  1. Constructs an AST
  2. Fills out some rudimentary information in a symbol table

    We use the State to maintain the program AST and a symbol table.

    ```python
    class State:
        def __init__(self):
            self.initialize()

        def initialize(self):
            # symbol table to hold variable-value associations
            self.symbol_table = {}

            # when done parsing this variable will hold our AST
            self.AST = None

    state = State()
    ```
Consider the rule: IF (' exp ') stmt opt_else
What does the tuple tree look like for the various shapes of the ‘if’ statement?
def p_prog(p):
   
   program : stmt_list
   
   state.AST = p[1]

Save the construct AST in the state!

def p_stmt_list(p):
   
   stmt_list : stmt stmt_list | empty
   
   if (len(p) == 3):
       p[0] = ('seq', p[1], p[2])
   elif (len(p) == 2):
       p[0] = p[1]

Statement lists are ‘nil’ terminated ‘seq’ terms.

def p_empty(p):
   
   empty :
   
   p[0] = ('nil',)
This should look familiar, same structure as for the expressions in exp1bytecode language.
Running the Frontend

```
In [16]:
from cuppal_frontend_gram import parser
from cuppal_lex import lexer
from cuppal_state import state
from grammar_stuff import dump_AST

In [17]:
state.initialize()
parser.parse("get x; put x", lexer=lexer)

In [18]:
dump_AST(state.AST)

(seq
  |(get x)
  |(seq
  |(put
  |  |(id x))
  |(nil)))

In [19]:
state.symbol_table

Out[19]: {'x': None}
```
Running the Frontend

In [20]:
state.initialize()
parser.parse("get x; x = x + 1; put x", lexer=lexer)

In [21]:
dump_AST(state.AST)

(seq
  | (get x)
  | (seq
    | (assign x
    |   | (+
    |   |   | (id x)
    |   |   | (integer 1)))
    | (seq
    |   | (put
    |   |   | (id x))
    |   | (nil))))

In [22]:
state.symbol_table

Out[22]: {'x': None}
Running the Frontend

```python
In [23]:
    state.initialize()
    parser.parse("while (1) {}", lexer=lexer)

In [24]:
dump_AST(state.AST)

(seq
   |(while
   |  |(integer 1)
   |  |(block
   |   |  |(nil)))
  |(nil))
```
Running the Frontend

In [25]:
    state.initialize()
    parser.parse("get x; if (0 <= x) put 1", lexer=lexer)

In [26]:
dump_AST(state.AST)

(seq
  |(get x)
  |(seq
    |(if
      |(<=
        |(|(integer 0)
        |(|(id x))
      |(|(put
        |(|(integer 1))
        |(|(nil)))
    |(|(nil)))
  |(|(nil)))
Running the Frontend

In [27]: `parser.parse("get x; if (0 <= x) put 1 else put 2", lexer=lexer)`

In [28]: `dump_AST(state.AST)`

```
(seq
 |(get x)
 |(seq
   |(if
     |(<=
     |(integer 0)
     |(id x))
   |(put
     |(integer 1))
   |(put
     |(integer 2)))
 |(nil)))
```
Processing ASTs: Tree Walking

- The recursive structure of trees gives rise to an elegant way of processing trees: *tree walking*.

- A tree walker typically starts at the root node and traverses the tree in a depth first manner.
Processing ASTs: Tree Walking

Consider the following:

```
In [34]: ast = ('+', ('*', ('integer', 3), ('integer', 2)), ('integer', 4))
```

```
In [35]: from grammar_stuff import dump_AST
dump_AST(ast)
```

```
(+
 | (*
 | | (integer 3)
 | | (integer 2))
 | (integer 4))
```
Processing ASTs:

```python
def const(node):
    # pattern match the constant node
    (INTEGER, val) = node
    # return the value as an integer value
    return int(val)

def add(node):
    # pattern match the tree node
    (ADD, left, right) = node
    # recursively call the walker on the children
    left_val = walk(left)
    right_val = walk(right)
    # return the sum of the values of the children
    return left_val + right_val

def mult(node):
    # pattern match the tree node
    (MULT, left, right) = node
    # recursively call the walker on the children
    left_val = walk(left)
    right_val = walk(right)
    # return the product of the values of the children
    return left_val * right_val

def walk(node):
    # first component of any tree node is its type
    t = node[0]
    # lookup the function for this node
    node_function = dispatch_dict[t]
    # now call this function on our node and capture the return value
    val = node_function(node)
    return val
```

dispatch_dict = {
    '+' : add,
    '*' : mult,
    'integer' : const
}
Processing ASTs: Tree Walking

A simple tree walker for our expression tree

In [34]:

ast = ('+', ('*', ('integer', 3), ('integer', 2)), ('integer', 4))

In [35]:

from grammar_stuff import dump_AST
dump_AST(ast)

(+
 |(*
 | |(integer 3)
 | |(integer 2))
 |(integer 4))

In [39]:

print(walk(ast))

10

We just interpreted the expression tree!!!
Processing ASTs: Tree Walking

- Notice that this scheme mimics what we did in the syntax directed interpretation schema,

- But now we interpret an expression tree rather than the implicit tree constructed by the parser.

```python
def const(node):
    # pattern match the constant node
    (INTEGER, val) = node

    # return the value as an integer value
    return int(val)

def add(node):
    # pattern match the tree node
    (ADD, left, right) = node

    # recursively call the walker on the children
    left_val = walk(left)
    right_val = walk(right)

    # return the sum of the values of the children
    return left_val + right_val

def mult(node):
    # pattern match the tree node
    (MULT, left, right) = node

    # recursively call the walker on the children
    left_val = walk(left)
    right_val = walk(right)

    # return the product of the values of the children
    return left_val * right_val
```
Tree Walkers are Plug'n Play

- Tree walkers exist completely separately from the AST.
- Tree walkers plug into the AST and process it using their node functions.
Tree Walkers are Plug'n Play

- There is nothing to prevent us from plugging in multiple walkers during the processing of an AST, each performing a distinct phase of the processing.
An Interpreter for Cuppa1

Diagram:
- Cuppa1 Frontend
- Input
- build
- AST
- write
- Output
- Tree Walker for the Cuppa1 Interpreter
def walk(node):
    # node format: (TYPE, [child1[, child2[, ...]]])
    type = node[0]

    if type in dispatch_dict:
        node_function = dispatch_dict[type]
        return node_function(node)
    else:
        raise ValueError("walk: unknown tree node type: " + type)

# a dictionary to associate tree nodes with node functions
dispatch_dict = {
    'seq'    : seq,
    'nil'    : nil,
    'assign' : assign_stmt,
    'get'    : get_stmt,
    'put'    : put_stmt,
    'while'  : while_stmt,
    'if'     : if_stmt,
    'block'  : block_stmt,
    'integer': integer_exp,
    'id'     : id_exp,
    'paren'  : paren_exp,
    '+'      : plus_exp,
    '-'      : minus_exp,
    '*'      : times_exp,
    '/'      : divide_exp,
    '=='     : eq_exp,
    '<='     : le_exp,
    'uminus' : uminus_exp,
    'not'    : not_exp
}
def assign_stmt(node):
    (ASSIGN, name, exp) = node
    assert_match(ASSIGN, 'assign')
    value = walk(exp)
    state.symbol_table[name] = value

def seq(node):
    (SEQ, stmt, stmt_list) = node
    assert_match(SEQ, 'seq')
    walk(stmt)
    walk(stmt_list)

def while_stmt(node):
    (WHILE, cond, body) = node
    assert_match(WHILE, 'while')
    value = walk(cond)
    while value != 0:
        walk(body)
        value = walk(cond)

def if_stmt(node):
    try: # try the if-then pattern
        (IF, cond, then_stmt, (NIL,)) = node
        assert_match(IF, 'if')
        assert_match(NIL, 'nil')
        value = walk(cond)
        if value != 0:
            walk(then_stmt)
        else:
            walk(else_stmt)
        return
    except ValueError: # if-then pattern didn't match
        (IF, cond, then_stmt, else_stmt) = node
        assert_match(IF, 'if')
        value = walk(cond)
        if value != 0:
            walk(then_stmt)
        else:
            walk(else_stmt)
        return

def plus_exp(node):
    (PLUS, c1, c2) = node
    assert_match(PLUS, '+')
    v1 = walk(c1)
    v2 = walk(c2)
    return v1 + v2

An Interpreter for Cuppa1

cuppa1_interp_walk.py
An Interpreter for Cuppa1

```python
from argparse import ArgumentParser
from cuppa_lex import lexer
from cuppa_frontend_gram import parser
from cuppa_state import state
from cuppa_interp_walk import walk

def interp(input_stream):
    # initialize the state object
    state.initialize()

    # build the AST
    parser.parse(input_stream, lexer=lexer)

    # walk the AST
    walk(state.AST)

if __name__ == "__main__":
    # parse command line args
    aparser = ArgumentParser()
    aparser.add_argument('input')

    args = vars(aparser.parse_args())

    f = open(args['input'], 'r')
    input_stream = f.read()
    f.close()

    # execute interpreter
    interp(input_stream=input_stream)
```

```
cuppa1_interp.py

In [49]: interp("get x; x = x + 1; put x")
Value for x? 3
> 4

In [50]: from cuppa1_examples import *

In [51]: print(list)

    // list of integers
    get x
    while (1 <= x)
    {
        put x;
        x = x + - 1;
        i = x
    }

In [52]: interp(list)
Value for x? 5
> 5
> 4
> 3
> 2
> 1
```
A Pretty Printer with a Twist

- Our pretty printer will do the following things:
  - It will read the Cuppa1 programs and construct an AST
  - It will compute whether a particular variable is used in the program
  - It will output a pretty printed version of the input script but will flag assignment/get statements to variables which are not used in the program

→ This cannot be accomplished in a syntax directed manner – therefore we need the AST
PrettyPrinting the Language

We need an IR because usage will always occur after definition – cannot be handled by a syntax directed pretty printer.
The Pretty Printer is a Translator!

- The Pretty Printer with a Twist fits neatly into our translator class
  - Read input file and construct AST/Collect info
  - Generate output code, flagging unused assignments
Pretty Printer Architecture

Frontend + 2 Tree Walkers
PP1: Variable Usage

- The first pass of the pretty printer walks the AST and looks for variables in expressions
  - only those count as usage points.
- A peek at the tree walker for the first pass, `cuppa1_pp1_walk.py` shows that it literally just walks the tree doing nothing until it finds a variable in an expression.
- If it finds a variable in an expression then the node function for `id_exp` marks the variable in the symbol table as used,
def assign_stmt(node):
    (ASSIGN, name, exp) = node
    assert_match(ASSIGN, 'assign')
    walk(exp)

def while_stmt(node):
    (WHILE, cond, body) = node
    assert_match(WHILE, 'while')
    walk(cond)
    walk(body)

def integer_exp(node):
    (INTEGER, value) = node
    assert_match(INTEGER, 'integer')

def binop_exp(node):
    (OP, c1, c2) = node
    if OP not in ['+', '-', '*', '/', '==', '<=']:
        raise ValueError("pattern match failed on " + OP)
    walk(c1)
    walk(c2)
PP1: Variable Usage

But...

```python
def id_exp(node):
    (ID, name) = node
    assert_match(ID, 'id')

    # we found a use scenario of a variable, if the variable is defined
    # set it to true
    if name in state.symbol_table:
        state.symbol_table[name] = True
```
Recall that when the frontend finds a definition of a variable as an assignment statement or a get statement, it enters the variable into the symbol table and initializes it with None.
**PP1: Variable Usage**

```
In [86]: from cuppal_frontend_gram import parser
   : from cuppal_lex import lexer
   : from cuppal_ppl_walk import walk as ppl_walk
   : from cuppal_state import state
   : state.initialize()

In [87]: parser.parse("get x", lexer=lexer)

In [88]: ppl_walk(state.AST)

In [89]: state.symbol_table

Out[89]: {'x': None}
```

Testing the tree walker

```
In [90]: state.initialize()

In [91]: parser.parse("get x; put x", lexer=lexer)

In [92]: ppl_walk(state.AST)

In [93]: state.symbol_table

Out[93]: {'x': True}
```
The tree walker for the second pass walks the AST and compiles a formatted string that represents the pretty printed program.

```python
def seq(node):
    (SEQ, s1, s2) = node
    assert_match(SEQ, 'seq')
    stmt = walk(s1)
    list = walk(s2)
    return stmt + list
```

Recall that programs are nil terminated Seq lists of statements:

```python
('seq',
    <Stmt1>,
    ('seq',
        <Stmt2>,
        ('nil',)))
```

Concatenate the string for stmt with the string from the rest of the Seq list.
def assign_stmt(node):
    (ASSIGN, name, exp) = node
    assert_match(ASSIGN, 'assign')
    exp_code = walk(exp)
    code = indent() + name + ' = ' + exp_code
    if not state.symbol_table[name]:
        code += ' // *** '+'name + ' is not used ***'
    code += '
    return code

def binop_exp(node):
    (OP, c1, c2) = node
    if OP not in ['+', '-', '*', '/', '==', '<=']:
        raise ValueError("pattern match failed on " + OP)
    lcode = walk(c1)
    rcode = walk(c2)
    code = lcode + ' ' + OP + ' ' + rcode
    return code

def while_stmt(node):
    global indent_level
    (WHILE, cond, body) = node
    assert_match(WHILE, 'while')
    cond_code = walk(cond)
    indent_level += 1
    body_code = walk(body)
    indent_level -= 1
    code = indent() + 'while (' + cond_code + ')
    return code

Indent() and indent_level keep track of the code indentation for formatting purposes.
Top Level Function of PP

```python
#!/usr/bin/env python
# Cuppa1 pretty printer

from sys import stdin
from cuppa1_frontend_gram import parser
from cuppa1_lex import lexer
from cuppa1_state import state
from cuppa1_pp1_walk import walk as pp1_walk
from cuppa1_pp2_walk import walk as pp2_walk
from cuppa1_pp2_walk import init_indent_level

def pp(input_stream = None):
    # if no input stream was given read from stdin
    if not input_stream:
        input_stream = stdin.read()

    # initialize the state object and indent level
    state.initialize()
    init_indent_level()

    # build the AST
    parser.parse(input_stream, lexer=lexer)

    # walk the AST
    pp1_walk(state.AST)
    code = pp2_walk(state.AST)

    # output the pretty printed code
    print(code)

if __name__ == "__main__":
    # execute only if run as a script
    pp()
```
The Cuppa1 PP

Testing the pretty printer

In [79]: from cuppa1_pp import pp

In [80]: pp("get x; while (1 <= x) { put x; x = x + -1; i = x }")

get x
while (1 <= x)
{
    put x
    x = x + -1
    i = x // *** i is not used ***
}
Assignment

- Assignment #5 – see webpage.