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 is not possible to design such a simple IR
- Instead, we use Abstract Syntax Trees (ASTs)

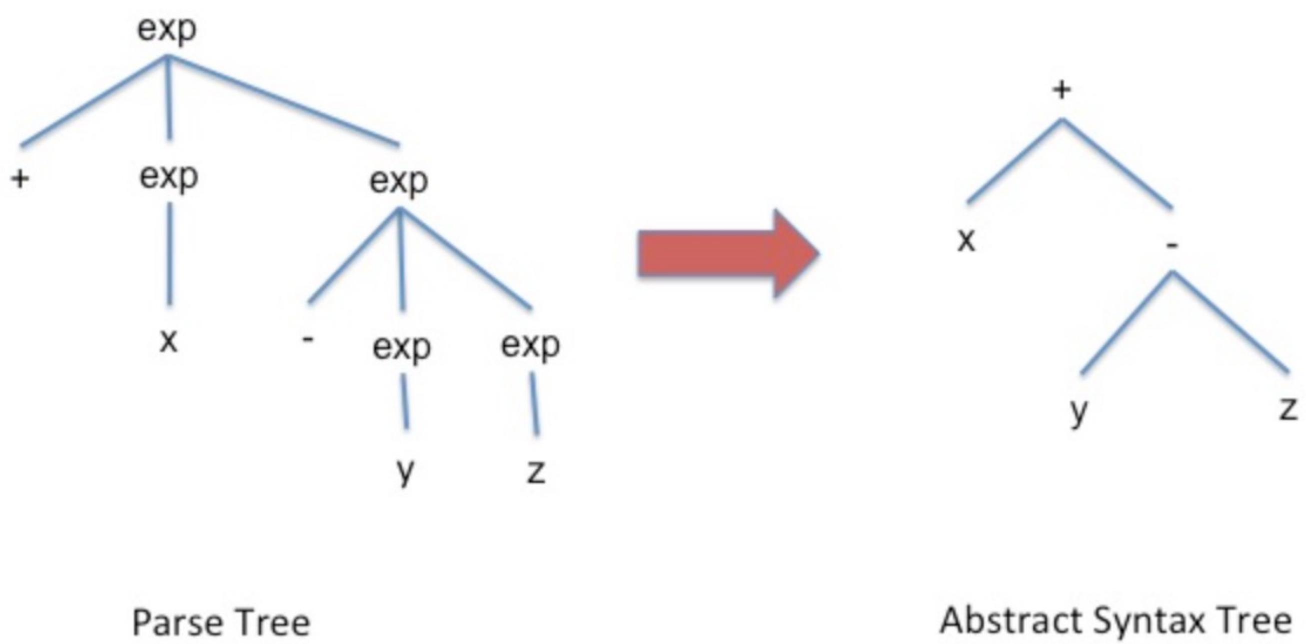
Chap 5

Reading

- Chap 5

Abstract Syntax Trees

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

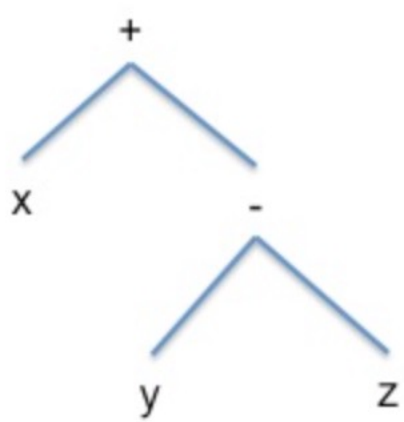


Abstract Syntax Trees

- 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

Tuple Representation of ASTs

- A convenient way to represent AST nodes is with the following structure,
  - (TYPE [, 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,




```
$ python3
>>> from dumpast import dumpast
>>> ast = ('+','x',('-','y','z'))
>>> dumpast(ast)

(+ x
  |(- y z))
>>>
```

The dumpast function will become your best friend!

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

Listing 5.1: A non-LL(1) Grammar for the Cuppa1 language.

```
stmt_list : (stmt)*

stmt : ID = exp ;?
     | get ID ;?
     | put exp ;?
     | while \( exp \) stmt
     | if \( exp \) stmt (else stmt)?
     | \{ stmt_list \}

exp : exp binop exp
    | primary

binop : + | - | \* | / | == | =<

primary : INTEGER
        | ID
        | \( exp \)
        | - primary
        | not primary

ID : <any valid variable name>
INTEGER : <any valid integer number>
```

Infix Expressions!

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

Problem: No precedence levels given!

The Cuppa1 Language

- Without precedence levels it is possible to create incorrect parse trees
- Solution: “Precedence Climbing”
  - Partition operators into precedence classes
  - Write grammar rules for each precedence class starting with the lowest operator precedence class.

The Cuppa1 Language

Listing 5.2: LL(1) grammar for the Cuppa1 language with precedence levels.

```
stmt_list : (stmt)*

stmt : ID = exp ;?
     | get ID ;?
     | put exp ;?
     | while \( exp \) stmt
     | if \( exp \) stmt (else stmt)?
     | \{ stmt_list \}

exp : exp_low
exp_low : exp_med ((== | =<) exp_med)*
exp_med : exp_high ((+ | -) exp_high)*
exp_high : primary ((\* | /) primary)*

primary : INTEGER
        | ID
        | \( exp \)
        | - primary
        | not primary

ID : <any valid variable name>
INTEGER : <any valid integer number>
```

Observe the rules for the operators: instead of the standard recursive term structure rules we treat operator expressions as LISTS of syntactic units

The prerequisite **left-associativity** of the operators comes naturally because LL(1) parsers execute the rule bodies from left to right.

The Lexer

```
token_specs = [
#   type:           value:
    ('COMMENT',     r'//.*'),
    ('GET',         r'get'),
    ('PUT',         r'put'),
    ('WHILE',       r'while'),
    ('IF',          r'if'),
    ('ELSE',        r'else'),
    ('NOT',         r'not'),
    ('ID',          r'[a-zA-Z][a-zA-Z0-9_]*'),
    ('INTEGER',     r'[0-9]+'),
    ('PLUS',        r'\+'),
    ('MINUS',       r'-'),
    ('MUL',         r'\*'),
    ('DIV',         r'/'),
    ('EQ',          r'=='),
    ('LE',          r'=<'),
    ('ASSIGN',      r'='),
    ('LPAREN',      r'\('),
    ('RPAREN',      r'\)'),
    ('LCURLY',      r'{'),
    ('RCURLY',      r'}'),
    ('SEMI',        r';'),
    ('WHITESPACE', r'[ \t\n]+'),
    ('UNKNOWN',     r'.'),
]
```

Tokenized Grammar & Lookahead Sets

```
stmt_list : ({ID,GET,PUT,WHILE,IF,LCURLY} stmt)*

stmt : {ID} ID ASSIGN exp ({SEMI} SEMI)?
     | {GET} GET ID ({SEMI} SEMI)?
     | {PUT} PUT exp ({SEMI} SEMI)?
     | {WHILE} WHILE LPAREN exp RPAREN stmt
     | {IF} IF LPAREN exp RPAREN stmt ({ELSE} ELSE stmt)?
     | {LCURLY} LCURLY stmt_list RCURLY

exp : {INTEGER,ID,LPAREN,MINUS,NOT} exp_low

exp_low : {INTEGER,ID,LPAREN,MINUS,NOT} exp_med
                    ({EQ,LE} (EQ|LE) exp_med)*

exp_med : {INTEGER,ID,LPAREN,MINUS,NOT} exp_high
                    ({PLUS,MINUS} (PLUS|MINUS) exp_high)*

exp_high : {INTEGER,ID,LPAREN,MINUS,NOT} primary
                    ({MUL,DIV} (MUL|DIV) primary)*

primary : {INTEGER} INTEGER
        | {ID} ID
        | {LPAREN} LPAREN exp RPAREN
        | {MINUS} MINUS primary
        | {NOT} NOT primary
```

We build the corresponding LL(1) parser in the usual fashion.

The Cuppa1 Frontend

- A frontend is a parser that constructs an AST
- Each parsing function returns a snippet of AST

cuppa1_fe.py

AST: Statements

```
stmt : {ID} ID ASSIGN exp ({SEMI} SEMI)?
```

The corresponding code snippet in the frontend is,

```
if token.type in ['ID']:
    id_tk = stream.match('ID')
    stream.match('ASSIGN')
    e = exp(stream)
    if stream.pointer().type in ['SEMI']:
        stream.match('SEMI')
    return ('ASSIGN', ('ID', id_tk.value), e)
```

```
stmt : {WHILE} WHILE LPAREN exp RPAREN stmt
```

This is implemented in the frontend as,

```
elif token.type in ['WHILE']:
    stream.match('WHILE')
    stream.match('LPAREN')
    e = exp(stream)
    stream.match('RPAREN')
    s = stmt(stream)
    return ('WHILE', e, s)
```

```
stmt : {IF} IF LPAREN exp RPAREN stmt ({ELSE} ELSE stmt)?
```

If-statements are interesting because part of the statements themselves are optional as indicated in the grammar rule with the question mark operator. The grammar rule is implemented by the frontend as,

```
elif token.type in ['IF']:
    stream.match('IF')
    stream.match('LPAREN')
    e = exp(stream)
    stream.match(RPAREN)
    s1 = stmt(stream)
    if stream.pointer().type in ['ELSE']:
        stream.match('ELSE')
        s2 = stmt(stream)
        return ('IF', e, s1, s2)
    else:
        return ('IF', e, s1, ('NIL',))
```

cuppa1_fe.py

AST: Statement Lists

```
stmt_list : ({ID,GET,PUT,WHILE,IF,LCURLY} stmt)*
```

and implemented in the frontend with the function,

```
def stmt_list(stream):
    lst = []
    while stream.pointer().type in ['ID','GET','PUT','WHILE','IF','LCURLY']:
        s = stmt(stream)
        lst.append(s)
    return ('STMTLIST', lst)
```

AST: Expressions

cuppa1_fe.py

```
primary : {INTEGER} INTEGER
```

The frontend implementation for this is,

```
if stream.pointer().type in ['INTEGER']:
    tk = stream.match('INTEGER')
    return ('INTEGER', int(tk.value))
```

This should look familiar, similar structure as for the expressions in exp1bytecode language.

```
elif stream.pointer().type in ['ID']:
        tk = stream.match('ID')
        return ('ID', tk.value)
```

```
primary : {MINUS} MINUS primary
```

The corresponding frontend code is,

```
elif stream.pointer().type in ['MINUS']:
    stream.match('MINUS')
    e = primary(stream)
    if e[0] == 'INTEGER':
        return ('INTEGER', -int(e[1]))
    else:
        return ('UMINUS', e)
```

Next we look at the medium precedence PLUS and MINUS operati
corresponding grammar rule is,

```
exp_med : {INTEGER,ID,LPAREN,MINUS,NOT} exp_high
                    ({PLUS,MINUS} (PLUS|MINUS) exp_high)*
```

```
def exp_med(stream):
    if stream.pointer().type in ['INTEGER','ID','LPAREN','MINUS','NOT']:
        e = exp_high(stream)
        while stream.pointer().type in ['PLUS','MINUS']:
            if stream.pointer().type == 'PLUS':
                op_tk = stream.match('PLUS')
            else:
                op_tk = stream.match('MINUS')
            tmp = exp_high(stream)
            e = (op_tk.type, e, tmp)
        return e
    else:
        raise SyntaxError("exp_med: syntax error at {}"
                          .format(stream.pointer().value))
```

Running the Frontend

```
$ python3
>>> from cuppa1_fe import parse
>>> from dumpast import dumpast
>>> ast = parse("get x; x = x + 1; put x;")
>>> dumpast(ast)

(STMTLIST
  |[
  |   |(GET
  |   |   |(ID x))
  |   |(ASSIGN
  |   |   |(ID x)
  |   |   |(PLUS
  |   |   |   |(ID x)
  |   |   |   |(INTEGER 1)))
  |   |(PUT
  |   |   |(ID x))])
>>>
```

```
$ python3
>>> from cuppa1_fe import parse
>>> from dumpast import dumpast
>>> ast = parse("while (1) {}")
>>> dumpast(ast)

(STMTLIST
  |[
  |   |(WHILE
  |   |   |(INTEGER 1)
  |   |   |(BLOCK
  |   |   |   |(STMTLIST
  |   |   |   |   |[]))))])
>>>
```

Running the Frontend

```
$ python3
>>> from cuppa1_fe import parse
>>> from dumpast import dumpast

### parse the program
>>> ast = parse("get x; if (0 =< x) put 1;")
>>> dumpast(ast)

(STMTLIST
  |[
  |   |(GET
  |   |   |(ID x))
  |   |(IF
  |   |   |(LE
  |   |   |   |(INTEGER 0)
  |   |   |   |(ID x))
  |   |   |(PUT
  |   |   |   |(INTEGER 1))
  |   |   |(NIL))])
>>>
```

```
$ python3
>>> from cuppa1_fe import parse
>>> from dumpast import dumpast

### parse the program
>>> ast = parse("get x; if (0 =< x) put 1 else put 2;")
>>> dumpast(ast)

(STMTLIST
  |[
  |   |(GET
  |   |   |(ID x))
  |   |(IF
  |   |   |(LE
  |   |   |   |(INTEGER 0)
  |   |   |   |(ID x))
  |   |   |(PUT
  |   |   |   |(INTEGER 1))
  |   |   |(PUT
  |   |   |   |(INTEGER 2)))])
>>>
```