CSCI: 4500/6500 Programming Languages

Conclusion of Lex and YACC and the Theory behind them (today– focus on YACC)

YACC Background

- *Review*: Recall grammars for YACC are a variant of BNF
  - » Can be used to express context free languages
    X -> p
  - » X is non terminal, p is a string of non-terminals and/or terminals)
  - » Context *free* because X can be replaced by p regardless of the *context* that X is in.

Some YACC Theory in this Context

- YACC - reduces an 'expression' to a single non-terminal (the start symbol)
- Is a bottom up or 'shift-reduce' parser (LR – Parses Left to right, right-most).
  - » (L) Reads the string from left to right (like westerners) and (R) produces the right-most derivations.

Example: 'Generating' a String (not parsing a string – yet)

- Example: Grammar that multiply and adds numbers:
  - » E ➔ E + E (rule 1)
  - » E ➔ E * E (rule 2)
  - » E ➔ id (rule 3)
- id is returned by lex (returns terminals) and only appears on right hand side.
  - » x + y * z is generated by:

E ➔	E * E	(rule 2)
➔	E * z	(rule 3)
➔	E + E * z	(rule 1)
➔	E + y * z	(rule 3)
➔	x + y * z	(rule 3)

To Parse the Language we need to go in reverse of generating the grammar

Now – How YACC Parses.

E ➔ E + E (rule 1)
E ➔ E * E (rule 2)
E ➔ id (rule 3)

- To parse the expression we go in reverse, reduce an expression to a single non terminal, We do this by shift-reduce parsing and use a stack for storing the terms

```
 1) . x + y * z        shift (terms on stack are on the left of dot)
 2) x . + y * z        reduce (rule 3)
 3) E . + y * z        shift
 4) E + . y * z        shift
 5) E + y. * z         reduce (rule 3)
 6) E + E. * z         shift
 7) E + E * . z        shift
 8) E + E * z .        reduce (rule 3) emit multiply
 9) E + E * E .        reduce (rule 2) emit add
10) E + E .            reduce (rule 1)
11) E .                Accept
```

- When we have a match on the stack to one of right hand side of productions replace the match with the left hand side of token

A Conflict at Step 6 (Ambiguity)

E ➔ E + E (rule 1)
E ➔ E * E (rule 2)
E ➔ id (rule 3)

- To parse the expression we go in reverse, reduce an expression to a single non terminal, We do this by shift-reduce parsing and use a stack for storing terms

```
 1) . x + y * z        shift (stack on left of dot)
 2) x . + y * z        reduce (rule 3)
 3) E . + y * z        shift
 4) E + . y * z        shift
 5) E + y. * z         reduce (rule 3)
 6) E + E. * z         shift  (here it is choice – reduce 'E+E' or shift)
 7) E + E * . z        shift
 8) E + E * z .        reduce (rule 3) emit multiply
 9) E + E * E .        reduce (rule 2) emit add
10) E + E .            reduce (rule 1)
11) E .                Accept
```

- "shift reduce" conflict at step 6 ambiguous grammar

Ambiguity

Ambiguity means the parser can't decide what to do:

- **Shift-Reduce Conflict:**
  - » Can't decide whether to shift or reduce a handle to a non-terminal
- **Reduce-Reduce Conflict:**
  - » Can't decide whether to reduce to on or more non-terminal.
  - E ➔ T
  - E ➔ id
  - T ➔ id
  - » Either reduces to E or to T

Ambiguity

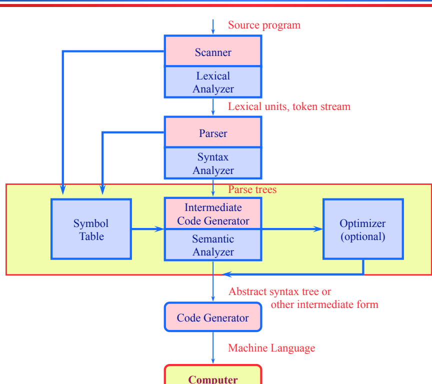
- This choice means we can't construct a *unique parse tree* for any string.
- But what if we could direct the parser to always prefer one choice over the other.
  - » Then
    - – The parse tree would always be unique
    - – The grammar might even be smaller
  - » How to resolve?
    - – Rewriting the grammar OR
    - – Indicate which operator has precedence (YACC enables this with the precedence definition)

Ambiguity: What Does YACC Do?

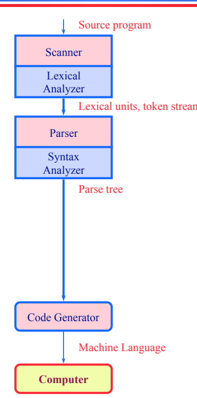
- Conflict Resolution Defaults:
  - » For shift-reduce conflicts YACC will always shift.
  - » For reduce-reduce conflict YACC selects the first rule.

- Reflecting where we are… and what we have done so far…
- Jflap

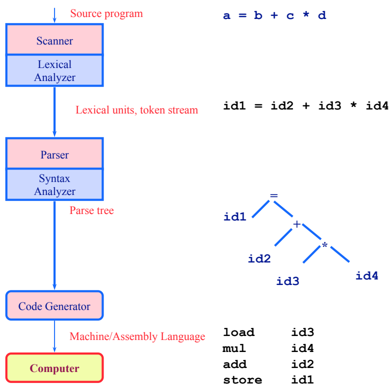
Big Picture: Compilation Process



Big Picture: Compilation Process



Big Picture: Compilation Process



Syntax: Regular Expressions (Tokens) & Context Free Grammars

- **Tokens: Described by regular expressions**
  - » First phase of compilation process converts strings/lexemes of the programming language to tokens (a representation of the lexeme in the computer)
    - – Example: letter (letter | digit) *
  - » Can be generated from just three rules/operations:
    - – Concatenation
    - – Repetition (arbitrary number of times - Kleene closure)
    - – Alternation (Choice from a finite set)
- **Context Free Language**
  - » Generated from 4 operations:
    - – Concatenation
    - – Repetition (arbitrary number of times - Kleene closure)
    - – Alternation (Choice from a finite set)
    - – Recursion

Definition of Languages

- **Recognizers**
  - » Reads input string and accepts or rejects if the string is in the language
  - » Example: Parsers -- the syntax analyzer of a compiler (yacc- yet another compiler compiler)
- **Generators**
  - » Generate sentences of a language
  - » Example: Grammars are language generators

Parse Trees

- Grammars describes 'hierarchical syntactic structures' so these can be "represented" by parse trees (e.g., a parser generates parse trees).
- Idea:
  - » To build a parse tree, put the start symbol at the root
  - » Add children to every non-terminal, following any one of the productions for that non-terminal in the grammar
  - » Done when all the leaves are tokens
  - » Read off leaves from left to right—that is the string derived by the tree

Example

```
Grammar:
<expr> ::= id | <number> | <expr> <op> <expr> | ( <expr> )
<op> ::= + | - | * | /
```

- **Generated String: slope * x + intercept**

Derivation and Sentenial form

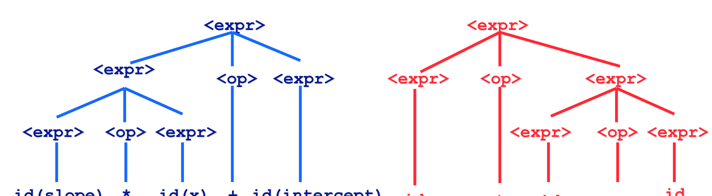
```
<expr>  ==>  <expr>  <op> <expr>
        ==>  <expr>  <op>  id
        ==>  <expr>   +    id
        ==>  <expr>  <op> <expr>  + id
        ==>  <expr>  <op>   id    + id
        ==>  <expr>   *     id    + id
        ==>   id      *     id    + id
            (slope)        (x)     (intercept)
```


Example

```
Grammar:
<expr> ::= id | <number> | <expr> <op> <expr> | ( <expr> )
<op> ::= + | - | * | /
```

```
<expr>  ==> <expr>  <op> <expr>
        ==> <expr>  <op>  id
        ==> <expr>   +    id
        ==> <expr>  <op> <expr> + id
        ==> <expr>  <op>  id    + id
        ==> <expr>   *    id    + id
        ==>  id      *    id    + id
```

```
<expr>  ==> <expr>  <op> <expr>
        ==> <expr>   *   <expr>
        ==>  id      *   <expr>
        ==>  id      *   <expr> <op> <expr>
        ==>  id      *   <expr>  +   <expr>
        ==>  id      *    id     +    id
```



Ambiguity

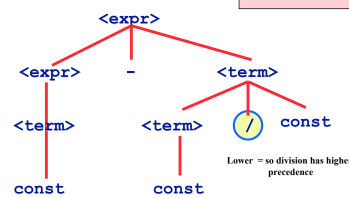
- The fact that some strings are the yield of more than one parse tree tells us that the grammar is ambiguous.
- Compiler often base the semantic on a phrase's parse tree
  - » More than one tree - cannot determine the meaning
    - – *Unless* there are some additional non-grammatical information
- Can include it in the grammar to facilitate the compiler to evaluate from the parse tree
- Precedence and associatively can be defined outside the grammar.

Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of operators we cannot have ambiguity

```
<expr> ::= <expr> <op> <expr> | const
<op> ::=  / | -
```

```
<expr> ::= <expr> - <term> | <term>
<term> ::=  <term> / const | const
```

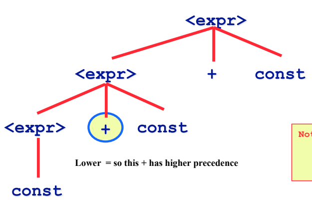


Hint: Higher precedence operators are lower in tree, here "/" has higher precedence than "-"

Associativity

- Operator associativity can also be indicated by a grammar
- Left Associative: 9+5+2 is equivalent to (9 + 5) + 2

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Note first addition is lower

2 Major Classes of Parsers

- LL - Left to right, left-most (discovers left most derivations – top down). *Predictive* parser.
  - » Works down the tree: left-right, predicting expanding nodes and tracking left most derivations.
- LR – (YACC) Left to right, right-most (discovers right most derivations). Bottom up parsers (e.g., Yacc - our focus).
  - » Notice a left is an ID next is a "," and then another ID. So it shifts until it can 'reduce'. Which doesn't happen until it sees a ';'.
- HW: See textbook (p. 63) for example on how these

```
<id-list> ::= id <id-list-tail>
<id-list-tail> ::= , id <id-list-tail>
<id-list-tail> ::= ;
```

```
A,B,C;
```


Context

- Programming languages require precise definitions (i.e., no ambiguity)
  - » Language *form* (Syntax)
  - » Language *meaning* (Semantics)
- Consequently, PLs are specified using formal notation:
  - » Formal syntax
    - – Tokens
    - – Grammar
  - » Formal semantics
    - – Static Semantics - Attribute Grammars (Compile Time)
    - – Dynamic Semantics (Run Time)

Static vs. Dynamic properties

- Static properties
  - » any property that may be determined through analysis of program text
    - – e.g., for some languages, the type of a program may be determined entirely through analysis of program source
      - e.g., ML, Java, & Pascal have "static type inference"
- Dynamic properties
  - » any property that may only be discovered through execution of the program
    - – e.g., "the final result of program p is 42" – may not be discovered without some form of execution
- Compilation involves forms of "static analysis"
  - » e.g., type checking, the definition and use of variables, information of data and control flow and much more.

Why Attribute Grammar?

- **Semantic Analyzer**: Analyses the "meaning" to Syntax.
- Enables **type compatibility checks** (e.g., float = int OK, int = float not OK) would require too many rules
- Enables Checking Declaring all variables before they are referenced can't be specified in BNF

Who?: Donald Knuth (father of the analysis of computer algorithms) designed Attribute Grammars to describe both syntax & static semantics (compile time)

What is an Attribute Grammar?

- Attribute Grammar = Context Free Grammar plus (+):
  - » Attributes (values assigned to grammar symbols)
  - » Attribute computation functions (how to compute attribute values)
  - » Predicate functions (static semantic rules)

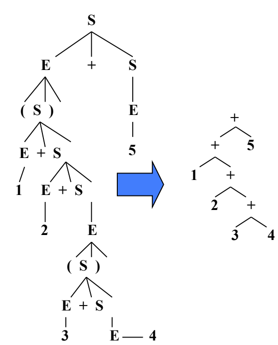
How ?

- Embellishes (decorates) the Context Free Grammar (Syntax) Tree, the parse tree:
  - » Annotates a simplified version (Abstract Syntax Tree) of the Syntax Tree (Concrete Syntax Tree).
    - Add values and semantics rules to grammar productions
    - Variable declared before they are declared
    - Type checking.

```
1.  During Parsing Create Tree
2.  Simplify Tree –and create Abstract Syntax Tree (AST)
3.  Annotate the AST
```

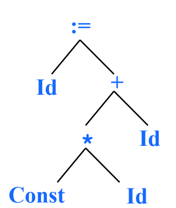

Abstract Syntax Tree (AST) - Review

- Derivation = sequence of applied productions
  - » S → E+S → 1+S → 1+E →1+2
- Parse tree = graph representation of a derivation
  - » Doesn't capture the order of applying the productions
- AST discards unnecessary information from the parse tree



Simple Example: Abstract Syntax Tree

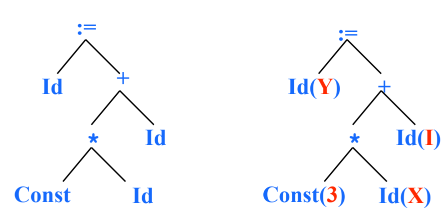
For "Y := 3 * X + I"



* such a tree could be produced by a compiler's "front end"

ASTs with "Attributes"

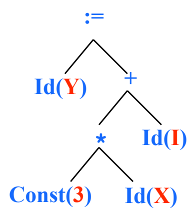
Attribute grammars are CFGs with extra information (a.k.a., "attributes") stored at the nodes



* red data are "initial attributes" in the lingo.

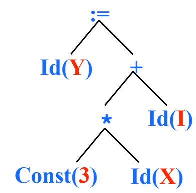
Attribute Grammars and Static Type checking

Assume: we know Y, I, and X are variables of type float
Question: is the following a legal program?



Attribute grammars and static checking

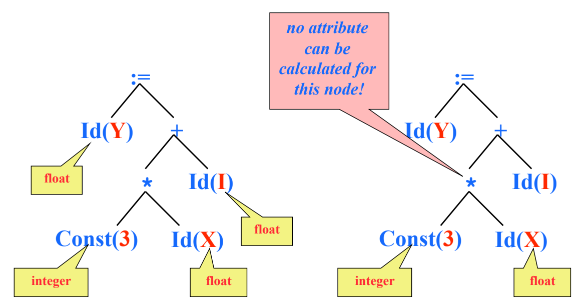
Assume: we know Y, I, and X are variables of type float
Question: is the following a legal program?



Answer: it depends on the language definition
- ML, Java, etc: no implicit coercion
- C, Basic, Scheme would allow

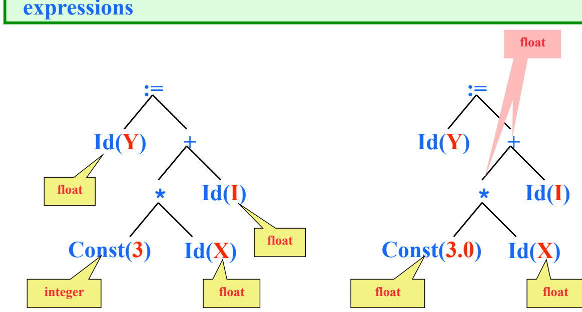
Attribute grammars and static checking

First Case (Java, ML): it's illegal

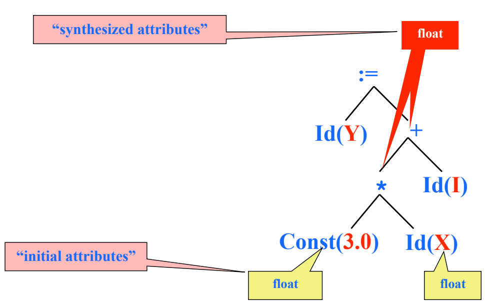


Attribute grammars and static checking

Second Case (C, Scheme): implicitly coerce the constant so that it makes sense; calculate the types of the intermediate expressions



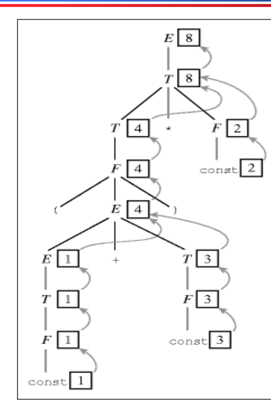
Attribute grammars and static checking



Attribute Flow

Example (Text Book p. 169)

- **The figure shows the result of annotating the parse tree for `(1+3)*2`**
- **Each symbols has at most one attribute shown in the corresponding box**
  - » Numerical value in this example
  - » Operator symbols have no value
- **Arrows represent the *attribute flow***



Copy Rules & Semantics Functions

1: $E_1 \longrightarrow E_2 + T$
▷ E_1.val := sum (E_2.val, T.val)

2: $E_1 \longrightarrow E_2 - T$
▷ E_1.val := difference (E_2.val, T.val)

3: $E \longrightarrow T$
▷ E.val := T.val

4: $T_1 \longrightarrow T_2 * F$
▷ T_1.val := product (T_2.val, F.val)

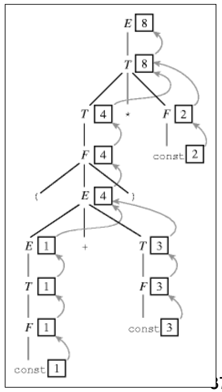
5: $T_1 \longrightarrow T_2 \; / \; F$
▷ T_1.val := quotient (T_2.val, F.val)

6: $T \longrightarrow F$
▷ T.val := F.val

7: $F_1 \longrightarrow - F_2$
▷ F_1.val := additive_inverse (F_2.val)

8: $F \longrightarrow (E)$
▷ F.val := E.val

9: $F \longrightarrow$ const
▷ F.val := const.val



Attribute Flow
Synthetic and Inherited Attributes

- **In the previous example, semantic information is pass up the parse tree**
  - » **We call this type of attributes are called *synthetic attributes***
  - » **Attribute grammar with synthetic attributes only are said to be *S-attributed***
- **Semantic information can also be passed down the parse tree**
  - » **Using *inherited attributes***
  - » **Attribute grammar with inherited attributes only are said to be *non-S-attributed***

HW: Reading

- **Chapters 1,2**
  - » **Derivations of Parse Trees**
  - » **Difference between Top DOWN and Bottom UP Parsing**
- **Sections: 4.1-4.4**
  - » **Semantic Analysis**
    - – **Dynamic, Static Checks**
    - – **Attribute Grammar**
    - – **Evaluating Attribute**
      - **Synthesized**
      - **Inherited**
      - **Attribute Flow**