CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107HS in the Winter 2015/2016 term at the University of Toronto

Reading Assignment

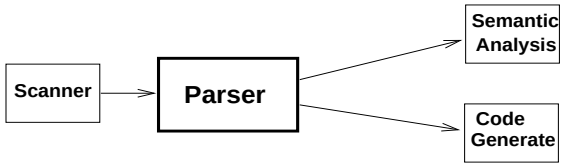
Section 10.1

Chapter 7

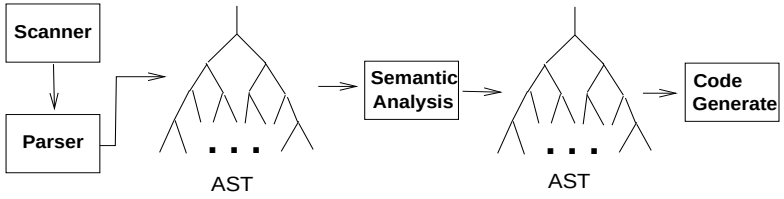
Omit 7.7

Emphasize AST use and structure

Single Pass Compiler



Multi Pass Compiler



Internal Compiler Data Structures

- Information flows between compiler passes
  - Representation(s) of the program
  - Tables
  - Error messages
  - Compiler flags
  - Source program coordinates.
- Form of communication may change as program is processed. A compiler may use multiple representations of a program.
- Use disk resident information for very large programs. Use memory resident information for better compiler speed.
- **Backward information flow should be avoided if possible.**

Intermediate Representations

- Represent the structure of the program being compiled
  - Declaration structure.
  - Scope structure.
  - Control flow structure.
  - Source code structure.
- Used to pass information between compiler passes
  - Compact representation desirable.
  - Should be efficient to read and write.
  - Provide utility to print intermediate language for compiler debugging.

Intermediate Representation Examples

Condensed Source

A * B + C / D - E

Polish Postfix Notation

A B * C D / + E -

Triples

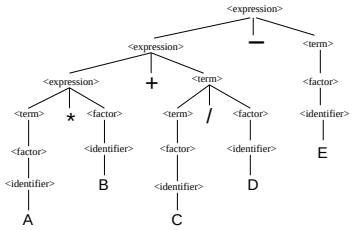
1 (* , A , B)
2 (/ , C , D)
3 (+ , (1) , (2))
4 (- , (3) , E)

Quadruples

(* , A , B , T1)
(/ , C , D , T2)
(+ , T1 , T2 , T3)
(- , T3 , E , T4)

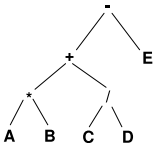
Parse Tree

Complete representation of the syntactic structure of the program according to some grammar.



Abstract Syntax Tree

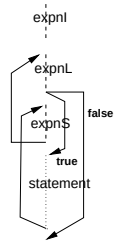
Similar to parse tree but only describes essential program structure.



Directed Acyclic Graphs

Used to represent control structure

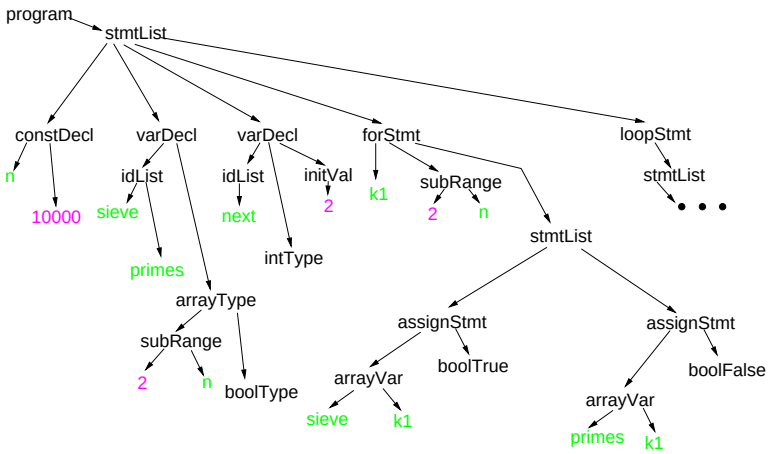
for (expnI ; expnL ; expnS **)** statement



Abstract Syntax Trees

- Abstract Syntax Trees (ASTs) are designed to capture all of the essential structural information about a program being compiled.
- Bottom Up syntax analysis is a good match with AST building. The child nodes to the tree are always built before the parent node.
- The basic process for *building* an AST is very simple:
  - the leaf nodes are typically constants and identifiers.
    Build a node to represent each of these entities.
  - Interior nodes are build as needed to represent particular language constructs.
    Typically interior nodes contain links to one or more child nodes.
- The *processing* of AST nodes for semantic analysis is also simple:
  - Process the AST *depth first*. This guarantees that child nodes are processed before parent nodes.
  - At each parent node do any processing required using information from already processed child nodes.

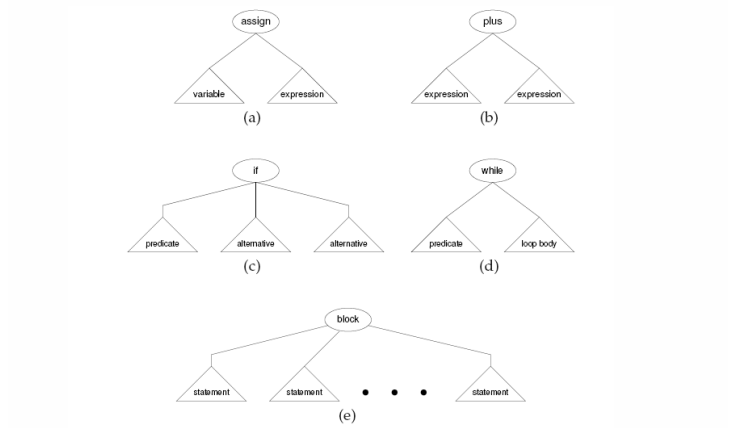
Partial[a] Abstract Syntax Tree for Running Example (Slide 25)



[a]Compare this to the full parse tree in Slide 58

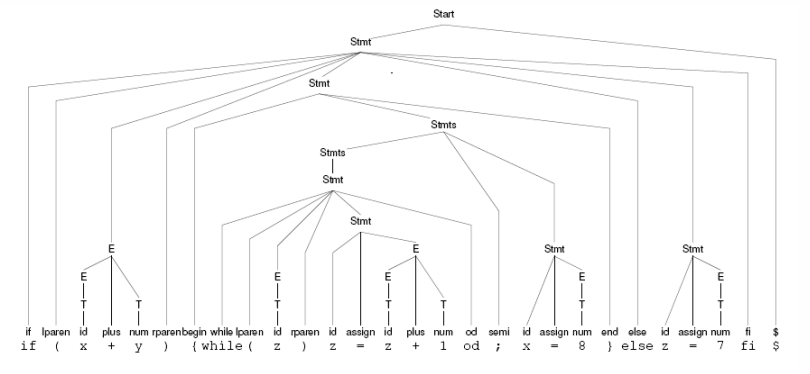
ASTs Are Language Specific

Typically AST notes are custom designed for each of the major constructs in the language. Some Examples (*Fischer, Cytron, LeBlanc* Figure 7-15)[a]:



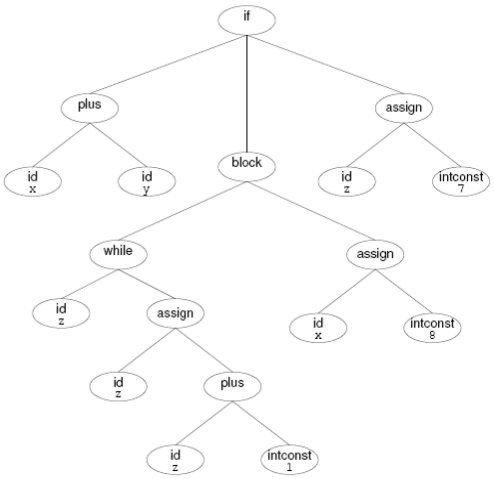
[a]For CSC488S Winter 2015/2016 examples see the AST classes: `AssignStmt`, `BinaryExpn`, `IfStmt`, `WhileDoStmt`, and `Scope`

Complete Syntax Tree for Program Fragment[a]



[a] *Fischer, Cytron, LeBlanc* Figure 7-18

Abstract Syntax Tree for Program Fragment[a]



[a] *Fischer, Cytron, LeBlanc* Figure 7-19

A Very General Abstract Syntax Tree Node[a]

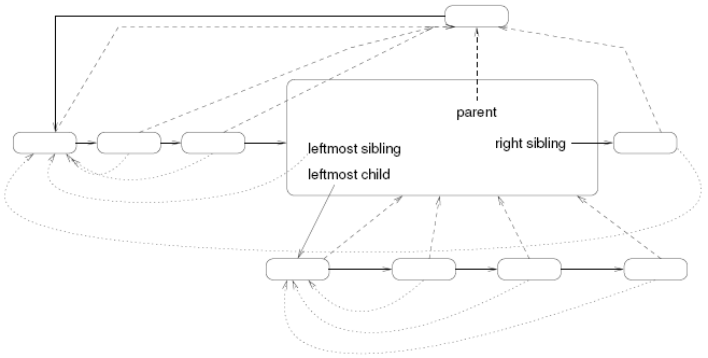


Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

[a] *Fischer, Cytron, LeBlanc* Figure 7-12