CMPT 379
Compilers

Anoop Sarkar
`http://www.cs.sfu.ca/~anoop`

Parse trees

- Given an input program, we convert the text into a parse tree
- Moving to the backend of the compiler: we will produce intermediate code from the parse tree
- This process is called syntax directed translation because we are using a CFG
- Parser output is a *concrete syntax tree*

Intermediate Representations

- A parse tree is an example of a very high level intermediate representation
- We can reconstruct the original source code from the concrete syntax tree
- Typically we want to check some semantic rules on the parse tree and report any errors
- The next step: semantic processing and code generation

Abstract Syntax Trees

- Take the concrete syntax tree and simplify it to the essential nodes
- For example, if the parser used an LL(1) grammar then the concrete syntax tree will have extra non-terminals
- Elimination of left-recursion, changing the grammar to remove shift/reduce conflicts

Abstract Syntax Trees

- Assume we have a top-down parser, e.g. an LL (1) parser.
- We have to eliminate left-recursion to use the parser

  $E \rightarrow E + T \mid T$

  Becomes

  $E \rightarrow T\ E_1$ and $E_1 \rightarrow +\ T\ E_1 \mid \varepsilon$
- For future steps, the AST might convert back into a tree that is compatible with the original grammar (before left-recursion elimination)

Abstract Syntax Trees

- Another example is the use of built-in functions, user-defined functions and operators
- In each case we have to call some code with a number of parameters
- Each case might have a separate syntax with different punctuation marks, e.g. () ;
- Punctuation marks are useful in language design but not useful when presenting a uniform tree for future analysis and code generation
- In an AST, all of these cases can be converted to a single tree format

Abstract Syntax Trees

- Other examples include lists of various kinds that involves recursion in CFGs:

```
Program → Function-List
Function-List → Function-Defn Function_List
              | Function-Defn
```

- The extra nodes created due to these grammar changes are not useful
- The extra nodes might make things non-local (inconvenient) for the semantic processing and code generation

Abstract Syntax Trees

- Process the concrete syntax tree and convert into a tree that is useful for semantic processing and code generation
- Note that ambiguity is no longer a problem: we already have the parse tree
- Abstract syntax trees will typically have pointers to children *and* pointers to parent nodes

Example

- Consider the following fragment of a programming language grammar:

  Program → Function-List

  Function-List → Function-Defn Function-List

  | Function-Defn

  Function-Defn → **fun id** (Param-List) Body

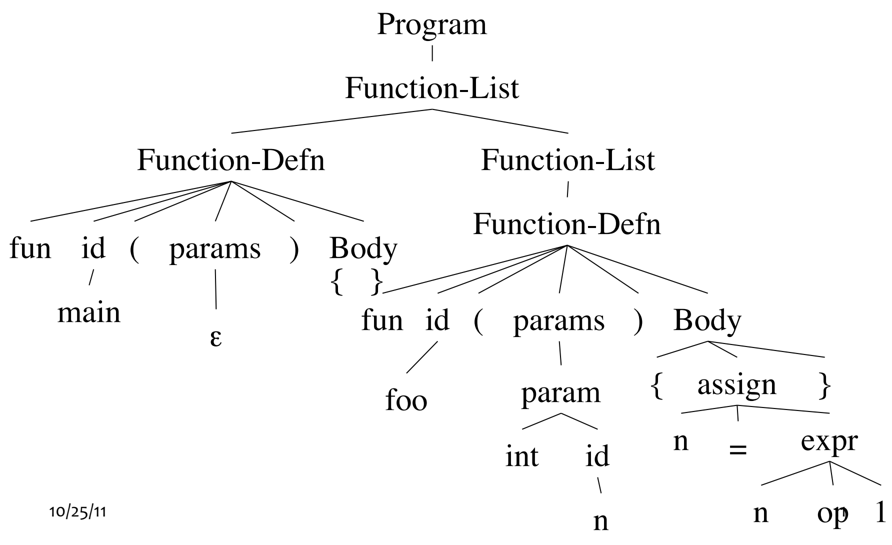
  Body → **'{'** Statement-List **'}'**

Example (cont'd)

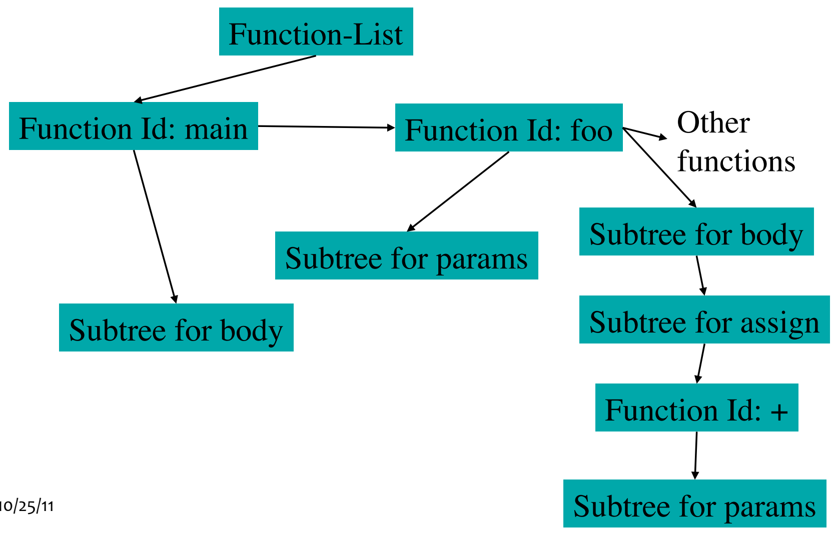
- Consider an example program:

```
fun main ()
{
    statement
}
fun foo (int n)
{
    n = n + 1
}
```

Concrete Parse Tree



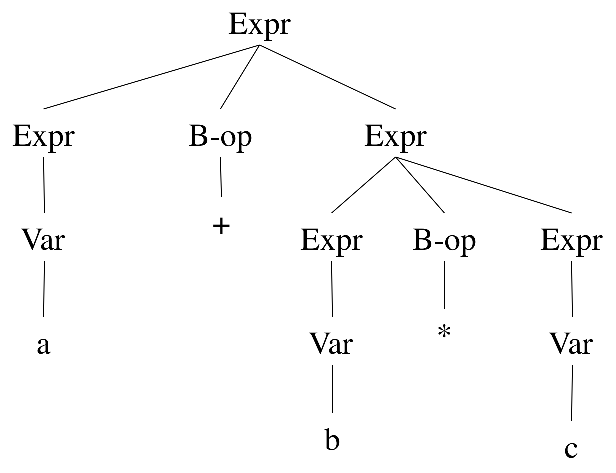
Abstract Parse Tree



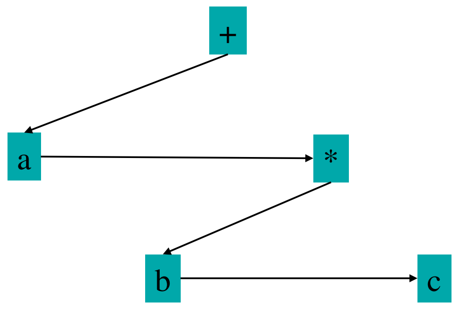
Code generation as Translation

- Code generation can be viewed as translation from the parse tree
- In other words, an alignment between the source code and the assembly code
- Typically we go to an intermediate representation and then to assembly
- Let's consider a simple case where the IR step can be skipped

Expr concrete syntax tree



Expr abstract parse tree



Code generation

- GenerateCode(tree t, int resultRegister)
- Recursively traverse the abstract syntax tree
- At each node produce the code needed for that binary operation based on the results from the recursive call results

Trace of code generation

```
GenerateCode(+, 0)
    GenerateCode(a, 0)
        Write "LOAD a, R0"
    GenerateCode(*, 1)
        GenerateCode(b, 1)
            Write "LOAD b, R1"
        GenerateCode(c, 2)
            Write "LOAD c, R2"
        Write "MUL R1, R2"
    Write "ADD R0, R1"
```

Result of code generation

- The resulting assembly code:

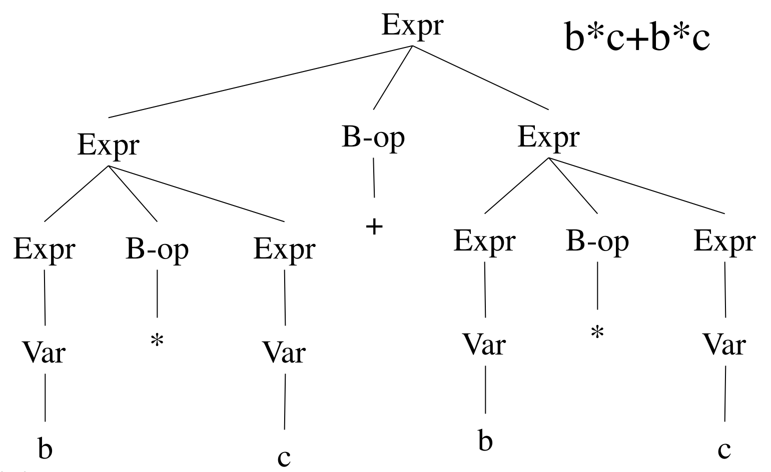
```
LOAD a, R0
LOAD b, R1
LOAD c, R2
MUL R1, R2
ADD R0, R1
```

- Note that using the tree structure means that the registers do not conflict
- Later we will consider the optimal assignment of values to registers

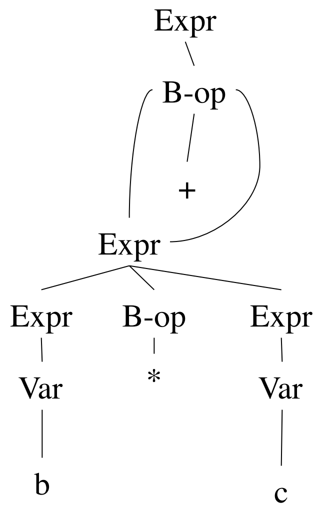
Case Study: Lisp

- The term abstract syntax was coined by John McCarthy
- McCarthy designed Lisp which directly used an abstract syntax bypassing the concrete syntax step
- Structure of Lisp: (*function arg-list*)
- Directly represents the parse tree in syntax
- Lisp: Lots of Irritating Silly Parentheses

Directed Acyclic Graphs



Directed Acyclic Graphs



Summary

- The parser produces concrete syntax trees
- Abstract syntax trees: abstract away from any grammar transformations or remove unnecessary punctuation
- Tree is input for code generation
- Ad-hoc code generation from ASTs
- As before, we would like to formally specify translation from AST to assembly/machine code
- ASTs can also be the basis for semantic analysis