Prototyping a Compiler

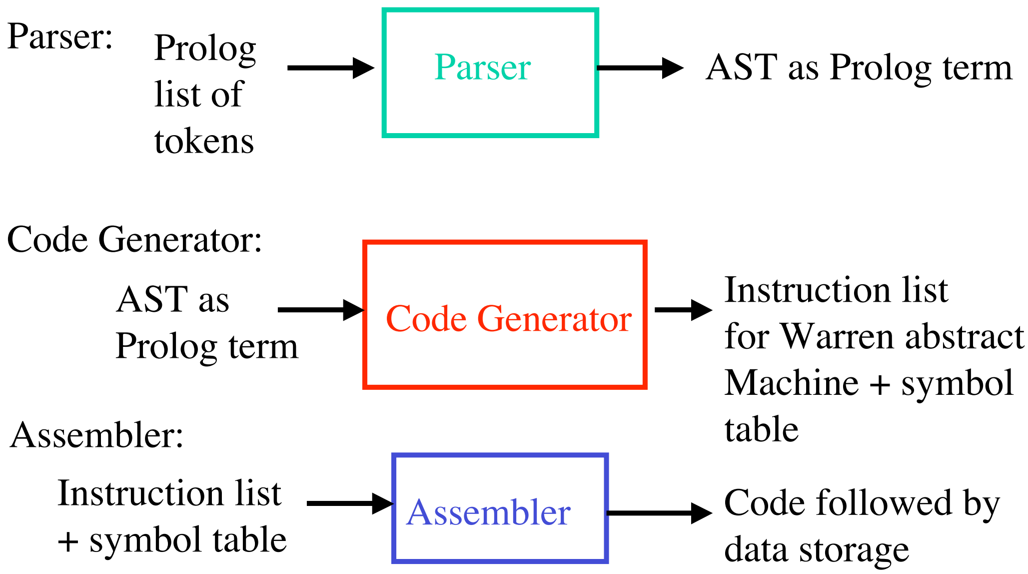
D.H.D. Warren, "Logic Programming and Compiler Writing", Software Practice and Experience, vol 10, pp 97-125, 1980.

- **Prolog features used**
  - **Logic variables with late binding**
  - **Unification**
  - **AST's built as Prolog terms**
- **Build recursive descent parser with code generation into a list**
- **Example: translating arithmetic expressions**

Prototype Compiler

- **Source: subset of Pascal/C**
- **Target: von Neumann machine code**
- **Claim:**
  - **Code is self-documenting (through choice of variable names)**
  - **Facilitates experiments in language design**
  - **Compiler design is very modular, built with TD design;**
    - **UNIX pipe-type communication between compiler phases;**
    - **Uses LL parsing**

Compiler



Compilation

- ***Lexical analysis*: provided input program splits input line into a flat Prolog list of tokens**
- ***Parsing*: create intermediate code (AST) from token stream**
- ***Code generation*: create basic structure of object program with symbolic addresses; build symbol table**

Compilation, cont.

- *Assembly*: map data to storage; fix up symbolic addresses to absolute addresses
- Consider each portion of the TD compiler in turn
- Input to be a token stream in a Prolog list
- Output to be a stream of instructions followed by data storage

Parsing

- Each nonterminal becomes a Prolog term with three arguments:

  <nonterm> (<start>, <end>, <tree>)

  where

  <start> is a token stream in a Prolog list,

  <end> is remaining token stream after <nonterm> is recognized,

  <tree> is top level of the AST corresponding to <nonterm>

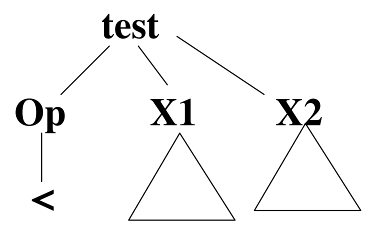
Parser Example

e.g., <stmt> ::= if <test> then <stmt> else <stmt> becomes

stmt([if | Z0], Z, if(Test,Then,Else)):-
  test(Z0, [then | Z1], Test), stmt(Z1, [else | Z2], Then),
  stmt(Z2, Z, Else).

test(Z0, Z, test(Op, X1, X2)):- expr(Z0, [Op | Z1], X1),
  compareop(Op), expr (Z1, Z, X2).

expr(Z0, Z, X) :- subexpr(… etc.



Note, our Prolog [X| Y] is equivalent to Warren's [X . Y]

Example - If Stmt

if x>0 then y:= 2 else y := 3 | Z2, Z1, Z0

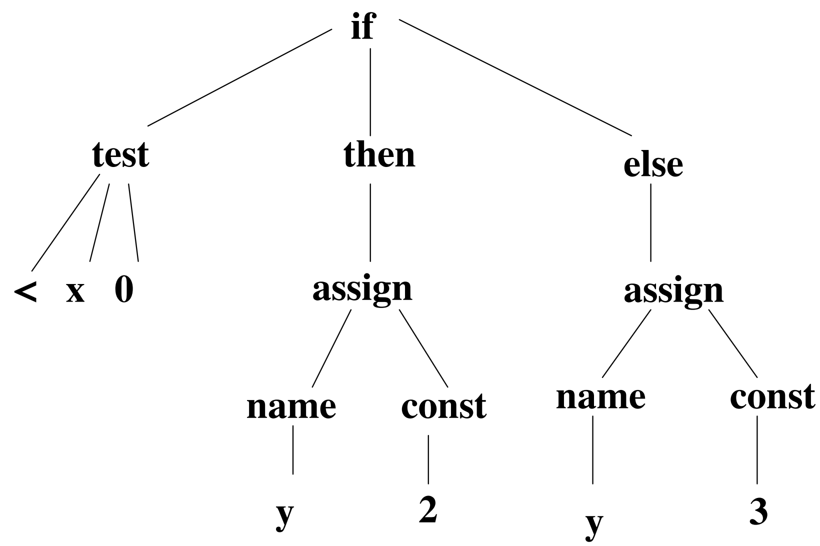
Warren, p 120

^ call to <stmt> unifies with [if |Z0] as start
  ^ call to <test>
    first call to <expr> to find x
    second call to <expr> to find 0
    returns test(> , x, 0) in <test> rule which matches "then"
      ^ call to stmt(Z1, [else | Z2], Then) finds first assignment, y:=2
        ^call to stmt(Z2, Z, Else) finds second assignment, y:=3

(^ shows approximate location in input stream)

Example - If Stmt AST

if(test(<,X,0), then(assign(name(y),const(2))), else(assign(name(y),const(3))))



Code Generation

- To produce basic structure of object program with machine addresses in symbolic form
- Done through a tree walk

  encodestmt(<1>, <2>, <3>)

  <1> is input AST constructed by parser

  <2> is dictionary, gives bindings for names, will eventually hold offset addresses

  <3> output object code

Code Generation Example

```
encodestmt(assign(name(X),Expr), D,
     (Exprcode; instr(store,Addr)) ):-
          lookup(X,D,Addr),
          encodeexpr(Expr,D,Exprcode).
```

encodestmt(AST for assignment stmt, dictionary or symbol table, (Code for rhs of assignment; code for the store instruct.)):-

Addr is address for X to be bound to actual storage later, during assembly

encodeexpr generates code for Expr AST with symbol table D

Warren, p 110

Code Generation Example

- Uses unification and delayed binding to generate code with "holes" for data addresses to be filled in later
- Actually, ordering of subgoal evaluation here is irrelevant
- Note: in paper, code is generated in infix format (a flat sequence) rather than the Prolog prefix form we're showing

  [instruct1; instruct2; instruct3; …]

Example

Warren, p 113

source: **if <test> then <stmt> else <stmt>**

object code:

	Testcode
	Thencode
	Jump <label2>
<label1>:	**Elsecode**
<label2>:	

has embedded jump to <label1> on false value

in code L1,L2 are unbound vars, whose values are set at assembly time; automatic handling of forward references!!

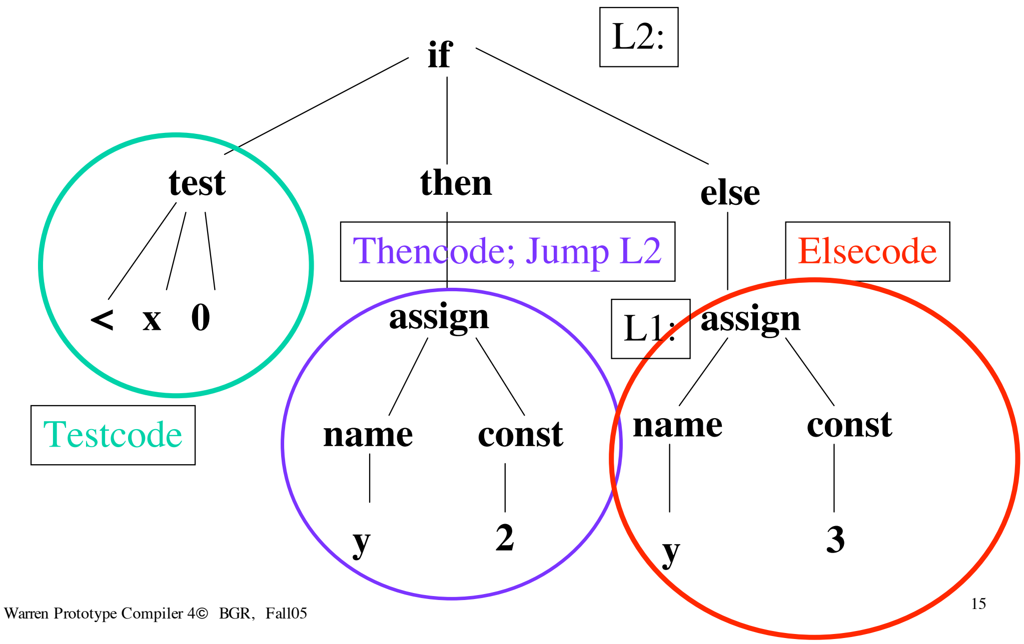
Example

```
encodestmt( if(Test,Then,Else), D, (Testcode;
   Thencode; instr(jump L2); label(L1);
   Elsecode; label(L2)) ) :-
      encodetest(Test, D, L1, Testcode),
      encodestmt(Then, D, Thencode),
      encodestmt(Else, D, Elsecode).
encodetest(test(Op,Arg1,Arg2), D, Label,
   (Exprcode ; instr(Jumpif,Label)) ) :-
   encodeexpr(expr(-, Arg1, Arg2), D, Exprcode),
   unlessop(Op, Jumpif).
```

placeholder for forward jump

picks proper operator for comparison op

Example



Warren Machine Code

for If Stmt example:

Load &x %found by lookup
Loadc 0
JumpLE L1
Loadc 2
Store #y
Jump L2
L1: Loadc 3
Store #y
L2:

All variable locations resolved to absolute locations at assembly time

Instruction Set (Table 1,p107)

ADDC	ADD	JumpEQ	Read
SUBC	SUB	JumpNE	Write
MULC	MUL	JumpGT	Halt
DIVC	DIV	JumpLT	Block
LOADC	LOAD	JumpLE	
	STORE	JumpGE	
		Jump	

Symbol Table

- ***Symbol table*** **is called a dictionary**
- ***Dictionary*** **- an ordered tree of (name,value) pairs**
- **lookup(<1>,<2>,<3>): name <1> with value <3> is in dictionary <2>**
- **lookup is used to create dictionary, insert values and then retrieve them**
  - **Code generator builds dictionary and uses it for lookups;**
  - **Assembler associates addresses with names.**

Symbol Table Example

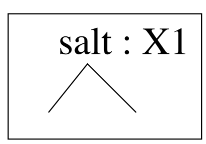
```
%find name, value at root
lookup(Name, dict(Name,Value, _ , _ ), Value):- !.
%look in left subtree
lookup(Name, dict(Name1, _ , Before, _ ), Value):-
        Name < Name1, lookup(Name,Before,Value).
%look in right subtree
lookup(Name, dict(Name1, _ , _ , After), Value):-
        Name > Name1, lookup(Name,After,Value).
```

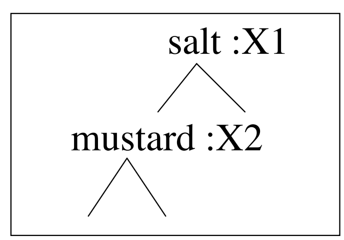
clause order for efficiency of evaluation

Table Building

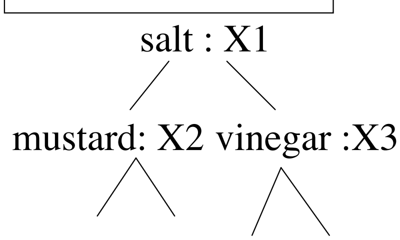
At first, D is empty.

lookup(salt, D, X1),



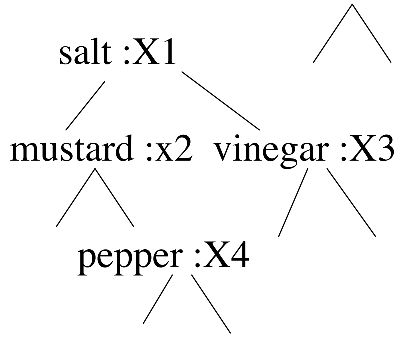


lookup(mustard, D, X2),



lookup(vinegar, D, X3),

lookup(pepper, D, X4).



Assembler

- **Names are resolved to absolute locations**
- **Labels bound to code locations**

compile(Source, (Code; instr(halt,0); block(L))):-

encodestmt(Source, D, Code*),%returns code and dictionary*

assemble(Code, 1, N0), *%computes addresses of labeled instructions and returns N0, end address of code*

N1 is N0 +1,

allocate(D, N1, N),*%lays out data storage from location N1 through N*

L is N - N1.*%length of data storage block*

Assembler

%N0 is code start address; N is code end address

assemble([Code1 | Code2], N0, N):-

assemble(Code1, N0, N1),

assemble(Code2, N1, N).

%increment instruction counter

assemble(instr(_, _), N0, N) :- N is N0 + 1.

%unifies location number with label

assemble(label(N), N,N).

Data Allocation

allocate(<1>, <2>, <3>) puts aside storage for all names in dictionary <1> between locations <2> and <3>.

allocate(void,N,N) :- !.*%choosing smallest dictionary*
allocate(dic(Name, N1, Before, After), N0, N):-
allocate(Before, N0, N1), N2 is N1+1,
allocate(After, N2, N).