Text Processing Tools

Alberto Ercolani

University of Trento

alberto.ercolani@unitn.it

November 30, 2017

Overview

Syntax Directed Translation

- Compilation is syntax driven: precedence and occurrence of symbols drive the execution of semantic actions. We saw it in lexers/parsers.
- The parser must produce an output for the next processing phase.

Syntax Directed Translation

- Def: A syntax directed translation is a grammar $\mathcal{G}$ augmented by semantic rules driving the translation process from a sequence of symbols to another medium (code/text).
- Given a production $\mathcal{P}$ of $\mathcal{G}$, the semantic rule (now called "translation rule") produces an output which is function of $\mathcal{P}$'s right hand side.
- Syntax directed translation is concerned with translation only, computations are a matter of SDDs.
- The easiest way to obtain the result is: building the annotated AST and visit it bottom up running the rules for each node.

Syntax Directed Translation, example

```
%% // Input: 31+42+53
E: E '+' T          {print('+');};
 | T                {/* Do nothing. */};
T: '(' E ')'        {/* Do nothing. */}
 | num              {print(num);print(" ")};
%%
/*
This SDT translates an infix arithmetic expression
to a postfix arithmetic expression.
*/
```

Syntax Directed Definitions

- Def: A syntax directed definition is a grammar $\mathcal{G}$ augmented by attributes and rules.

- Each non terminal symbol $\mathcal{N}$ is associated with a set of attributes which can be of two types.

    - Def: Synthesized attributes: their value depends on the values of the children of $\mathcal{N}$.

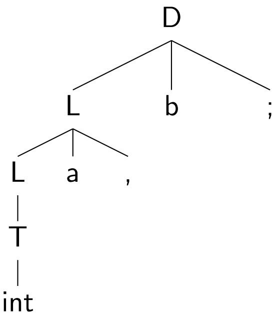
    - Def: Inherited attributes: their value depends on the values of $\mathcal{N}$'s parent **or** left siblings.

S-attributed grammars

- Def: S-attributed grammars, SDDs using synthesized attributes only.
- There are many simple examples of such grammars, SLR(1)/LALR(1)/LR(1) operator grammars.
- They naturally map to bottom up parsing.

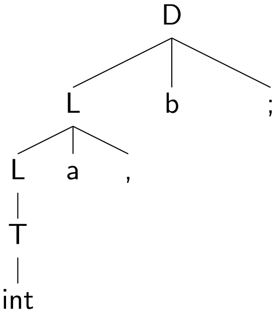
```
%%
E: E1 '+' T        {E.syn = E1.syn + T.syn;};
 | T               {E.syn = T.syn;};
T: T1 '*' F        {T.syn = T1.syn + F.syn;}
 | F               {T.syn = F.syn;}
F: '(' E ')'       {F.syn = E.syn;}
 | num             {F.syn = num.value;};
%%
```

Synthesized attributes, example




```
%% // Input: int a, b;
D: L id ';'        {id.type = L.type};
L: L1 id ','       {id.type=L1.type; L.type=L1.type}
| T                {L.type = T.type};
T: 'int'           {T.type = int}
| 'float'          {T.type = float};
%%
```

Synthesized attributes, example in Yacc



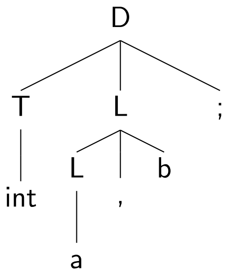

```
%% // Input: int a, b;
D: L id ';'        {id.type = $1};
L: L id ','        {id.type = $1; $$ = $1;}
| T                {$$ = $1};
T: 'int'           {$$ = int}
| 'float'          {$$ = float};
%%
```

L-attributed grammars

- Def: L-attributed grammars, SDDs using both synthesized and inherited attributes.
- There are many simple examples of such grammars, all S-attributed grammars and LL(1) operator grammars.
- They naturally map to top down parsing but they can not be evaluated during bottom up parsing. A real AST is required.

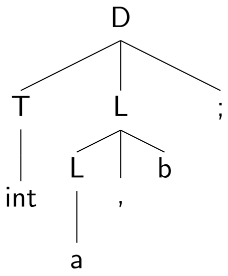
```
%%
D: T L ';'          {L.inh = T.syn};
T: 'int'            {T.syn = int}
 | 'float'          {T.syn = float};
L: L1 ',' id        {L1.inh = L.inh; id.type = L.inh}
 | id               {id.type = L.inh};
%%
```

Inherited attributes, example



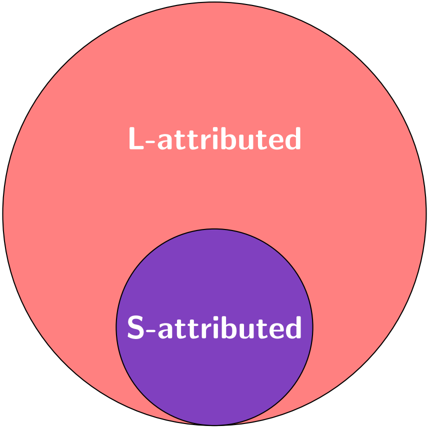

```
%% // Input: int a, b;
D: T L ';'        {L.inh = T.syn};
T: 'int'          {T.syn = int}
 | 'float'        {T.syn = float};
L: L1 ',' id      {L1.inh = L.inh; id.type = L.inh}
 | id             {id.type = L.inh};
%%
```

Inherited attributes, example in Yacc




```
%% // Input: int a, b;
D: T L ';'                    {/* Nope. */};
T: 'int'                      {$$ = INT;}
 | 'float'                    {$$ = FLOAT;};
L: L ',' id                   {$3.type = $0;}
 | id                         {$1.type = $0;};
%%
```

Relationship: S-attributed vs L-attributed grammars



Everything done by an S-attributed grammar can be done through an L-attributed grammar.

Syntax Directed Definition, some reminder

- Reminder: To visualize the computations carried out by SDDs, we assume to have the corresponding AST but its construction is not compulsory!
- Reminder: An S-attributed grammar can always be evaluated bottom up. (postorder traversal)
- Reminder: An SDD with mixed inherited and synthesized attributes doesn't necessarily have a valid ordering. (You can't evaluated them)

Some comment from the book

- Although inherited attributes can be useful, they can also be a source of **hard-to-find bugs**.
- An action that uses them has to take into account **every** place in the grammar where its rule is used.
- If you changed the grammar, you would have to make sure that, in the new place where the non terminal occurs, appropriate symbols precede it so that $0 will get the right value.

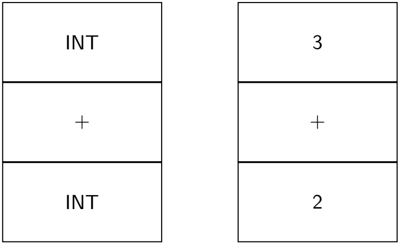
...

- It is usually safer and easier to use a global variable for the value that would have been fetched from a synthesized attribute.

The two stacks of Yacc

- Synthesized attributes are natural, because in Yacc, parsing happens in bottom-up fashion.
- Inherited attributes are unnatural during bottom up parsing: you must be aware of the operations carried out on the attribute stack.

The two stacks of Yacc



- During parsing explicit or implicit attribute handling is put in action.
  - Explicit: terminate semantic action with “$$=...”
  - Implicit: empty semantic action are filled with “$$=$1;”
  - Using the stack outside the *normal usage* (indexes lesser or equal 0) is hard.

Don't you believe me?

$$\emptyset$$

- $\underline{[\emptyset][\emptyset]}$
- ·int a, b;
- START: initially the tree, the symbol and attribute stack are empty.

Don't you believe me?

$$\emptyset$$

- $\underline{[\text{int}][\perp]}$
- ·int a, b;
- Lexer reads "int" and pushes a dummy value.

Don't you believe me?

$\emptyset$

- $\underline{[\text{int}][\perp]}$
- ·int a, b;
- SHIFT by "int"

Don't you believe me?

$$\emptyset$$

- $\underline{\text{[int][}\perp\text{]}}$
- int ·a, b;
- Lexer reads "a"

Don’t you believe me?

$\emptyset$

- $\underline{[\text{int}][\perp]}$
- int ·a, b;
- Parser reduces by rule “ T: 'int' {$$ = INT;}”

Don't you believe me?

$\emptyset$

- $\underline{\text{[int][}\perp\text{]}}$
- int ·a, b;
- Parser pops one element from symbol and attribute stack.

Don't you believe me?

$$\emptyset$$

- $\underline{[\emptyset][\emptyset]}$
- int ·a, b;
- Parser pops one element from symbol and attribute stack.

Don’t you believe me?

$$\emptyset$$

- $\underline{[\emptyset][\emptyset]}$
- int ·a, b;
- Parser pushes the driver of the production and value “INT”.

Don’t you believe me?

$\emptyset$

- [T][INT]
- int ·a, b;
- Parser pushes the driver of the production and value “INT”.
- After that the first couple of nodes is built.

Don't you believe me?

T
|
int

- [T][INT]
- int ·a, b;
- Lexer read "a", it is pushed on symbol stack, as before attribute is an undefined value.

Don't you believe me?

$$\begin{array}{c} \text{T} \\ | \\ \text{int} \end{array}$$

- [a][⊥]
- <u>[T][INT]</u>
- int ·a, b;
- SHIFT by "a".

Don't you believe me?

T

int

- [a][$\perp$]
- <u>[T][INT]</u>
- int a ·, b;
- Lexer reads ",".

Don’t you believe me?

$$\begin{array}{c} \text{T} \\ | \\ \text{int} \end{array}$$

- [a][$\perp$] \$1
- $\underline{\text{[T][INT]}}$ \$0
- int a$\cdot$, b;
- Parser reduces by rule “L: id $\{\$1.\text{type} = \$0;\}$”.

Don't you believe me?

$$\begin{array}{c} T \\ | \\ \text{int} \end{array}$$

- [a][⊥]
- [T][INT]
- int a·, b;
- Parser reduces by rule "L: id {$1.type = $0;}".

Don't you believe me?

T
|
int

- [a][⊥]
- <u>[T][INT]</u>
- int a·, b;
- Parser pops a symbol from symbol stack and its attribute value.

Don't you believe me?

T

int

- [T][INT]
- int a·, b;
- Parser pushes the driver and a dummy attribute value.

Don't you believe me?

T L

int a

- [L][⊥]
- [T][INT]
- int a·, b;
- The node couple is produced and parsing proceeds.
- Symbol "," is pushed and SHIFT by "," happens.

Don't you believe me?

$$
\begin{array}{ccc}
T & \qquad\qquad & L \\
| & & | \\
\text{int} & & a
\end{array}
$$

- [,][$\perp$]
- [L][$\perp$]
- $\underline{\text{[T][INT]}}$
- int a·, b;
- Symbol "," is pushed and SHIFT by "," happens.

Don’t you believe me?

$$\begin{array}{ccc} \mathsf{T} & \qquad\qquad & \mathsf{L} \\ | & & | \\ \mathsf{int} & & \mathsf{a} \end{array}$$

- [,][⊥]
- [L][⊥]
- <u>[T][INT]</u>
- int a,·b;
- Lexer reads “b”, SHIFT by “b”.

Don’t you believe me?

T	L
|	|
int	a

- [,][⊥]
- [L][⊥]
- <u>[T][INT]</u>
- int a,·b;
- “b” is pushed as tokens before.

Don't you believe me?

T　　　　L
|　　　　|
int　　　a

- [b][⊥]
- [,][⊥]
- [L][⊥]
- <u>[T][INT]</u>
- int a, b·;
- "b" is pushed as tokens before.

Don't you believe me?

```
T        L
|        |
int      a
```

- [b][⊥]
- [,][⊥]
- [L][⊥]
- [T][INT]
- int a, b·;
- Lexer reads ";".

Don't you believe me?

T　　　　L

int　　　　a

- [b][⊥]
- [,][⊥]
- [L][⊥]
- [T][INT]
- int a, b·;
- Reduction by rule "L: L ',' id {\$3.type = \$0;}".

Don't you believe me?

T　　　　　L
|　　　　　|
int　　　　a

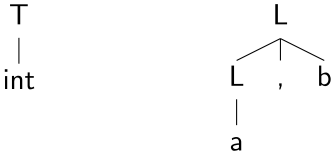
- [b][$\perp$] \$3
- [,][$\perp$] \$2
- [L][$\perp$] \$1
- $\underline{\text{[T][INT]}}$ \$0
- int a, b$\cdot$;
- Reduction by rule "L: L ',' id {\$3.type = \$0;}".

Don't you believe me?

T　　　　　　L
|　　　　　　|
int　　　　　a

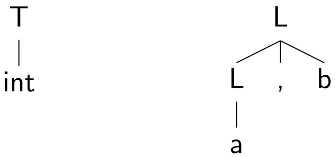
- [T][INT]
- int a, b·;
- Pop of 3 symbols on both stacks.

Don't you believe me?



- [L][⊥]
- <u>[T][INT]</u>
- int a, b·;
- Push of driver and construction of the tree.

Don’t you believe me?



- [L][⊥]
- [T][INT]
- int a, b·;

- I’m not showing the rest, my point was:
    - *You need to be careful, when using indexes i*≤ 0. *You* **must** *know what happened before in the stack.*
- If i used $-1 inside some rule, what should have happened according to you?

Don't you believe me?

- Inherited attributes are powerful but they are hard to manage in bottom up parsing.
- Yacc does not give you any correctness check about used indexes.

- Reminders:
  - In general using positive indexes is totally enough for our purposes. If you restrict your design to those, understanding the stack is optional.
  - Parsing and attributes stack are synchronized.
  - Once the reduction is complete $$ represents the top of the stack.

Summary

- Up to now we have been seeing some text processing tools.
  - A desk calculator, a basic imperative language interpreter, Make interpreter, a preprocessor.
  - These instances represent (SDT/SDD)s whose semantic actions are fired and input is elaborated on the fly.
- Serious processing tools like interpreters and compilers require complex data structures and sophisticated techniques.
- One pass evaluation is not enough.

Full Compilation Process

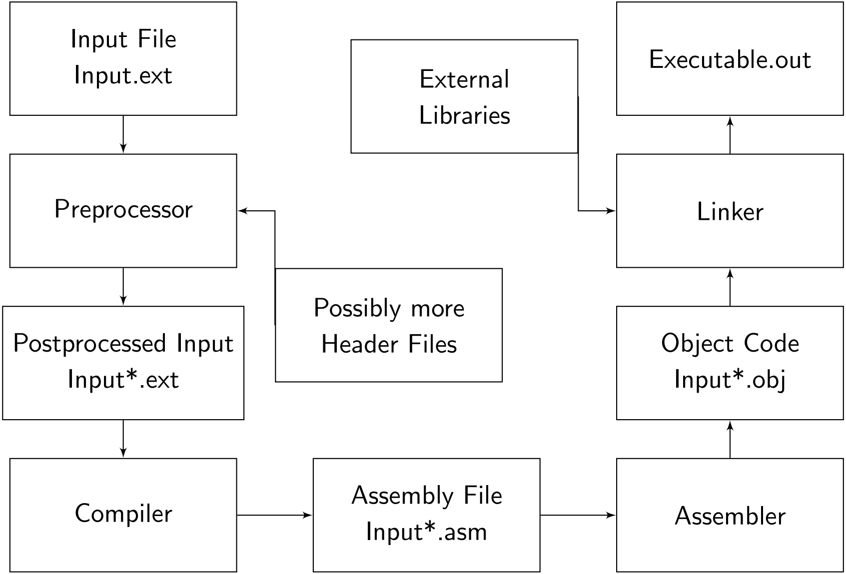
- Compilation is a complex process, implemented by a variety of tools.
- Compilers are divided into three parts: front, middle and back end.

- Front End: The stage checking syntactic, semantic correctness and building the *abstract syntax tree*.

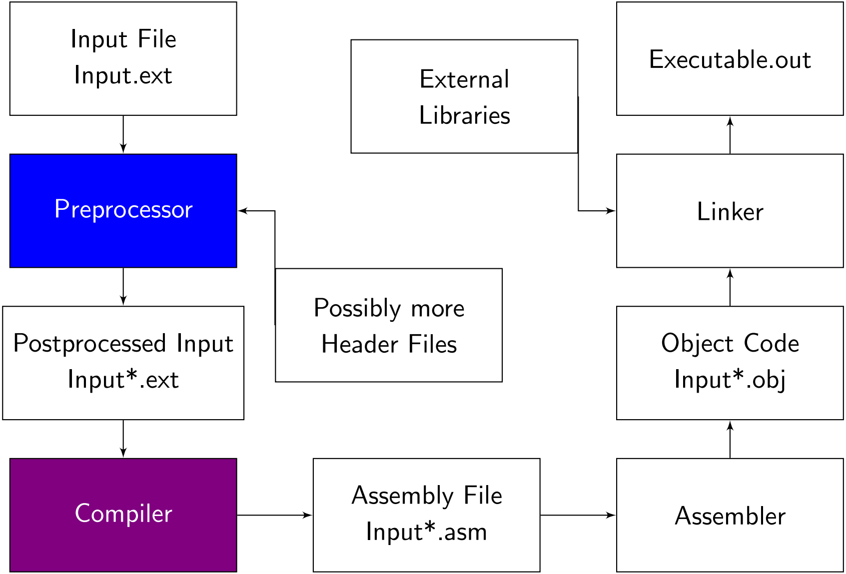
- Middle End: The stage where intermediate code is generated, transformed and optimized.

- Back End: The stage performing efficient translation to machine code.

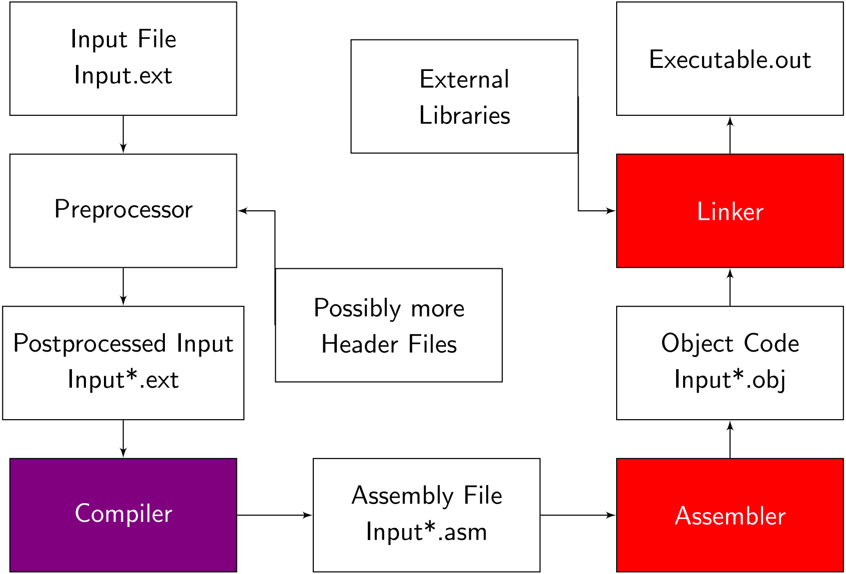
Full Compilation process



Front End Compilation



Back End Compilation



We consider front/middle ends only

- We will cover front and middle end without considering optimizations.
- Front End: The stage checking syntactic, semantic correctness, building the *abstract syntax tree*.
- Middle End: The stage where intermediate code is generated, transformed. ~~and optimized.~~
- ~~Back End: The stage performing efficient translation to machine code.~~

Syntactic/Semantic correctness

- Syntactic correctness is guaranteed by lexer and parser working together: we've been seeing this quite a lot.
- Semantic correctness is guaranteed by ad hoc functions running on specific data structure: we've been seeing it a little.
- What's left?
  - Abstract syntax tree
  - Type checking
  - Intermediate code generation

Abstract syntax tree

- An abstract syntax tree (AST) is a tree, whose *annotated* nodes have a variable number of children.
- On ASTs considerations about the input become simple because it gets structured and can be evaluated in any order.

- Yacc doesn't provide any integrated tool to build the AST, it's your duty.
- Moreover ASTs grow quickly, pruning procedures guarantee a reasonable size.

Growing an AST

$\emptyset$

- .5*2+3
- START

Growing an AST

$\emptyset$

- .5*2+3
- SHIFT

Growing an AST

$$\emptyset$$

- 5.*2+3
- SHIFT

Growing an AST

- 5.*2+3
- REDUCTION

Growing an AST

- 5*.2+3
- SHIFT

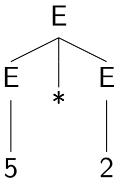
Growing an AST

- 5*2.+3
- SHIFT

Growing an AST

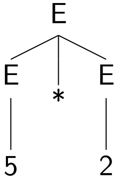
- 5*2.+3
- REDUCTION

Growing an AST



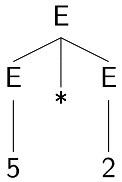
- 5*2.+3
- REDUCTION

Growing an AST



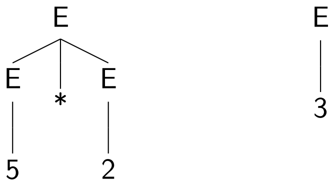
- 5*2+.3
- SHIFT

Growing an AST



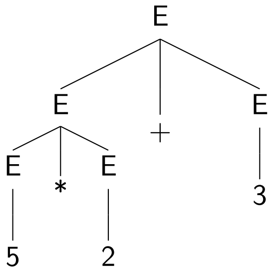
- 5*2+3.
- SHIFT

Growing an AST



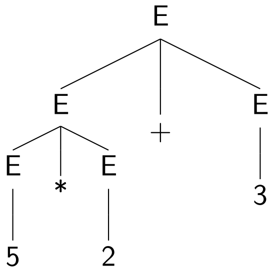
- 5*2+3.
- REDUCTION

Growing an AST



- 5*2+3.
- REDUCTION

Growing an AST



- 5*2+3.
- STOP
- For the first time, to visualize the AST we will consider it isomorph to the derivation tree.

Growing the real AST

$\emptyset$

- .5*2+3
- START

$\emptyset$

- .5*2+3
- SHIFT

Growing the real AST

$\emptyset$

- 5.*2+3
- SHIFT

Growing the real AST

5

- 5.*2+3
- REDUCTION

Growing the real AST

5

- 5*.2+3
- SHIFT

Growing the real AST

5

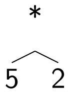
- 5*2.+3
- SHIFT

Growing the real AST

5 2

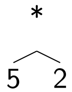
- 5*2.+3
- REDUCTION

Growing the real AST



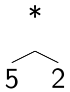
- 5*2.+3
- REDUCTION

Growing the real AST



- 5*2+.3
- SHIFT

Growing the real AST

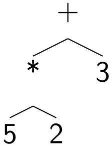


- 5*2+3.
- SHIFT

Growing the real AST

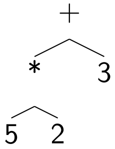
- 5*2+3.
- REDUCTION

Growing the real AST



- 5*2+3.
- REDUCTION

Growing the real AST



- 5*2+3.
- STOP
- The real AST encodes only strictly necessary data: data and operations.

Growing an AST: rules

```
%%
E:  E '+' E        {$$=MakeBOP($1, $3, SUM);}
  | E '-' E        {$$=MakeBOP($1, $3, DIFFERENCE);}
  | E '*' E        {$$=MakeBOP($1, $3, PRODUCT);}
  | E '/' E        {$$=MakeBOP($1, $3, DIVISION);}
  | '(' E ')'      {$$=MakeParenthesis($2);}
  | id             {$$=MakeIdentifier($1);}
  | num            {$$=MakeConstant($1);};
%%
```

- Assuming functions MakeConstant(·), MakeIdentifier(·), MakeParenthesis(·) and MakeBOP(·,·,·) produce nodes whose structure fields are correctly set the tree represents the original operation unambiguously.
- An AST isomorph to the corresponding derivation tree represent an unnecessary waste of memory and computational time.

AST Pruning

- Pruning operation discourages tree overgrowth through the employment of several strategies to be put in place:
  - Common subexpression elimination (sophisticated technique)
  - Constant nodes pre-evaluation (easy and cheap technique)
  - Singleton nodes pull up (a must have)
- We agreed on not talking about optimization.

AST Pruning: constant pre-evaluation

```
%%
E: E '+' E                {
      if ($1.IsConstant && $3.IsConstant){
              $$=MakeConstant($1.Value+$3.Value);
              free($1); free($3);
      }else{ $$=MakeBOP($1, $3, SUM); }
                          }
 | id                     {$$=MakeIdentifier($1);}
 | num                    {$$=MakeConstant($1);};
%%
```

- As you see, instead of adding new levels to the tree we delete old nodes and add new ones representing precomputed operations.
- By this policy we add levels only when compulsory.

AST Pruning: singleton node pull up

```
%%
E: E '+' E                    {$$=MakeBOP($1, $3, SUM);}
 | '(' E ')'                  {$$=$2;}
 | id                         {$$=MakeIdentifier($1);}
 | num                        {$$=MakeConstant($1);};
%%
```

- Production #2 is a good example of pull up.
- As you see, instead of adding new levels to the tree we retrieve singleton nodes (leaf or internal) and pass them to the father of the node.
- By this policy we add levels only when compulsory.

Type checking on the AST

- Type checking is important, it ensures the lack of unsafe operations:
  - Truncation (E.G.: long to int, different sizes)
  - Loss of precision (E.G.: float to int, same size different encoding)
- Type checking is extremely easy if carried out on the AST.
- Simply test the correctness of all operators considering involved types through "types tables".

Types tables

+	int	float	string	bool
int	int	float	string	$\perp$
float	float	float	string	$\perp$
string	string	string	string	string
bool	$\perp$	$\perp$	string	$\perp$

-	
int	int
float	float
string	$\perp$
bool	$\perp$

- Every operator has its own semantic. Sum between booleans and int or floats makes no sense in this case, C/C++ would let you sum them freely at your own risk.
- As you can notice the most precise type is always chosen. (float vs. int)
- "String" type is viral, every sum operation with it forces the output to be a string: this is called "type coercion".

Types tables: there also are unlucky operators

- According to you which (of the many) binary operator could behave like ♠?

♠	int	float	string	bool
int	bool	$\perp$	$\perp$	$\perp$
float	$\perp$	bool	$\perp$	$\perp$
string	$\perp$	$\perp$	bool	$\perp$
bool	$\perp$	$\perp$	$\perp$	bool

Growing a typed AST

$\emptyset, \perp$

- .5*2+3
- START

Growing a typed AST

$\emptyset, \perp$

- .5*2+3
- SHIFT

Growing a typed AST

$\emptyset, \perp$

- 5.*2+3
- SHIFT

Growing a typed AST

- 5.*2+3
- REDUCTION

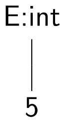
Growing a typed AST

E:int

5

- 5*.2+3
- SHIFT

Growing a typed AST

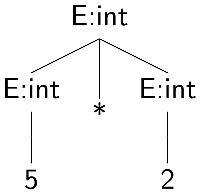


- 5*2.+3
- SHIFT

Growing a typed AST

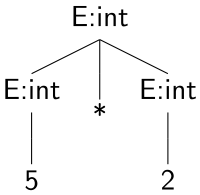
- 5*2.+3
- REDUCTION

Growing a typed AST



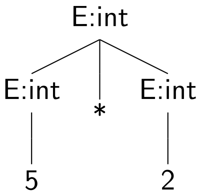
- 5*2.+3
- REDUCTION
- *Operator(int, int) = int; Ok!

Growing a typed AST



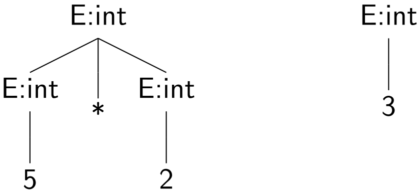
- 5*2+.3
- SHIFT

Growing a typed AST



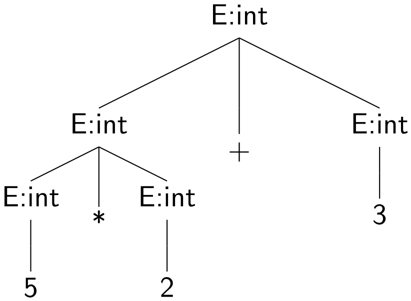
- 5*2+3.
- SHIFT

Growing a typed AST



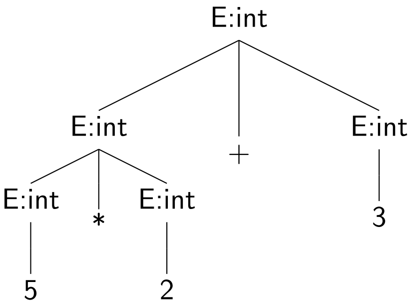
- 5*2+3.
- REDUCTION

Growing a typed AST



- 5*2+3.
- REDUCTION
- +Operator(int, int) = int; Ok!

Growing a typed AST

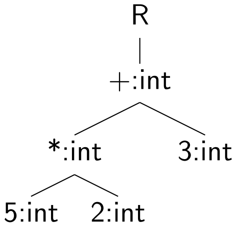


- 5*2+3.
- STOP
- Since the tree is typed we could evaluated the expression it represent allocating a variable whose type is the one of the root.
- The existence of the tree guarantees type correctness: in case of type mismatch, construction cancellation had to occur.

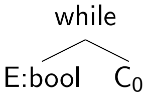
AST's role in interpreter's design

- AST is the simplest data structure allowing the design of an interpreter.
- Through a postorder traversal visit of the tree and the execution of specific pieces of code associated with each node, unambiguous execution/evaluation can be achieved.
- In the lab section of the moodle you will find packages: "Interpretation basics - boolean AST" and "Interpretation basics - Conditional Statements".

AST's role in interpreter's design

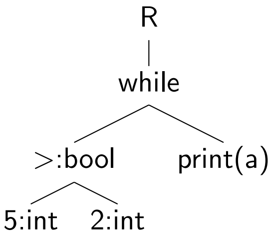


- Input = "5*2+3"
- Let R be the root of the tree, Evaluate(R) = 13;



- As seen in "imperative basics - conditional statements" package, the execution of a "while" node is driven by the value of expression E and in case it evaluates to true the execution of command C_0 can occur.

AST's role in interpreter's design



- Input = "while (5>2){print(a);}"
- Let R be the root of the tree, Execute(R) prints infinite "a"s and never halts.

Introduction to Intermediate Representation

- Up to now we saw how to build a simple interpreter (basic idea and flavor, can be generalized to any language), real compilation is missing.
- We won't dig too much into machine code generation just a look.
- A clever step in compiler design is the augmentation of the compiler by *intermediate representation* generation.

Intermediate Representation (I.R.)

- Def: A data structure or code used internally by a compiler or virtual machine to represent source code.
- In general I.R. is implemented by some code commonly referred to as "Intermediate Code" (I.C.).
- Once I.C. has been translated to object code (O.C.) it can be linked to target code (T.C.) and executed.

- This is not the only possible compilation chain, old compilers did output assembler code directly.

Intermediate Code generation

Producing intermediate code instead of producing assembler directly is a good idea: permits to develop a compiler having many possible target architectures.

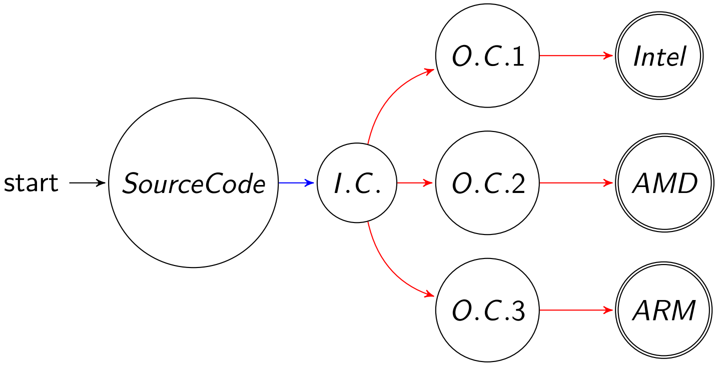


Figure: Blue arc: compiler's duty. Red arcs: assemblers' and linkers' duty

Intermediate Code has benefits

- A neutral representation with respect to the hardware.
- Easy to produce.
- Easy to inspect/read.
- Easier to optimize; optimization is machine independent.

Intermediate Code generation

- I.C. is often implemented through two different representations:
  - Postfix notation: the AST is linearized as a list of identifier references and operators.
  - Quadruples: all operations are transformed into tuples of four elements.

Intermediate Code generation: Postfix Notation

- Postfix notation is excellent for intermediate code generation because any expression can be written unambiguously without parenthesis and *without the need for precedence specification*.

- E.G.: Let input = "Exp = 2+(id*3)" be a valid assignment, then its postfix notation is:
- id 3 * 2 + Exp =

Postfix Notation generation: binary operators

- This is something we alredy saw in the context of SDTs.

```
%%
A: id '=' E        {print($1); print('=');}
E: E '+' T         {print('+');} | T         {};
T: T '*' F         {print('*');} | F         {};
F: '(' E ')'       {}
 | num             {print($1);print(" ")};
%%
/*Assuming id is associated to
a string and number to an integer.
Moreover parenthesis have no role as you can see.
*/
```

Running Postfix Notation

Let $\mathcal{E}$ be a postfix notation expression, e.g. $\mathcal{E} =$ "0 id 3 * 2 + -".

Algorithm 1: RunPostfixExpression

Input: $\mathcal{E}$ the postfix expression.

begin
- let $\mathcal{S}$ be a stack
- **foreach** *symbol* σ *in* $\mathcal{E}$ **do**
  - **if** $\neg$*IsOperator*(σ) **then**
    - $\mathcal{S}$.push(σ)
  - **else**
    - let α be the arity of operator σ
    - **for** *(i=0;i< α;i++)* **do**
      - pop and remember i-th top τ from $\mathcal{S}$
    - Compute the value $\rho = \sigma(\tau_0,\ldots,\tau_{\alpha-1})$
    - $\mathcal{S}$.push(ρ)

Postfix Notation generation: flow control

- Flow control has always been implemented through jump instructions (C *gotos*): opcodes whose purpose is changing the value of the program counter according to some condition.
- We can extend intermediate code postfix notation through a unary operator: **jump** label.
- Also conditional jumps can be encoded: **jump_if_false** & **jump_if_true**.

Postfix Notation generation: flow control

```
int iLabelIndex = 0, iElseStartLabel = 0,
    iElseEndLabel = 0;

%%
S: 'if' '(' E ')' M1 C0 M2 'else' C1 M3 {};
C: ··· /* Available commands· */

M1:     {iElseStartLabel = iLabelIndex;
        iLabelIndex++; print(iElseStartLabel);
        print('jump_if_false');};
M2:     {iElseEndLabel = iLabelIndex;
        iLabelIndex++;
        print(iElseEndLabel); print('jump');
        print(iElseStartLabel); printf(':');};
M3:     {print(iElseEndLabel);printf(':');};
%%
```

Flow Control: dissection

```
/* Three indexes used to get the next available
       label an to set the start and end point
       of the else case.
*/
int iLabelIndex = 0, iElseStartLabel = 0,
    iElseEndLabel = 0;
%%
...
%%
```

Flow Control: dissection

```
...
%%
S: 'if' '(' E ')' M1 C0 M2 'else' C1 M3 {};
C: ··· /* Available commands· */

/* We assume the grammar has expressions and
several commands; but most important, we
use three markers (M1, M2, M3) expanding
to  ϵ  what will print out the label numbers
and the instructions to jump· We also assume
non terminal C will produce some output
when reduced·*/
%%
```

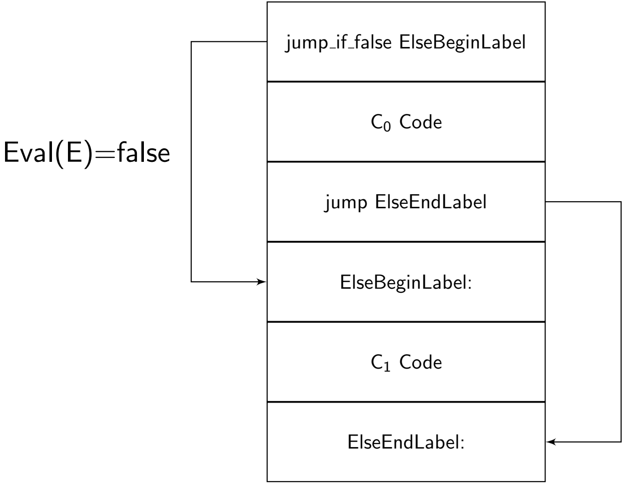
Flow Control: dissection

```
...
%%
M1:       {iElseStartLabel = iLabelIndex;
          iLabelIndex++; print(iElseStartLabel);
          print('jump_if_false');};

M2:       {iElseEndLabel = iLabelIndex;
          iLabelIndex++;
          print(iElseEndLabel); print('jump');
          print(iElseStartLabel); printf(':');};

M3:       {print(iElseEndLabel);printf(':');};
%%
```

Flow Control: dissection

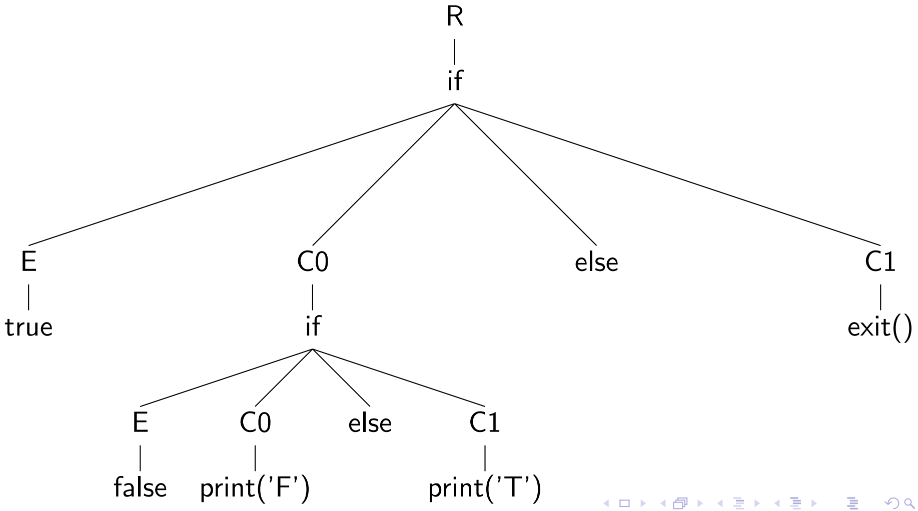


Flow Control: summary

- The trick lies in the reduction of every nonterminal from left to right.
- This specific design does not work inductively but gave you the idea.
- The easiest solution is memorizing the required labels in the AST node relative to the “if” command.
- Consider for instance, two nested “if” commands.
- All control flow instructions are based on condition evaluation: mastering “if” lets you master all other constructs.

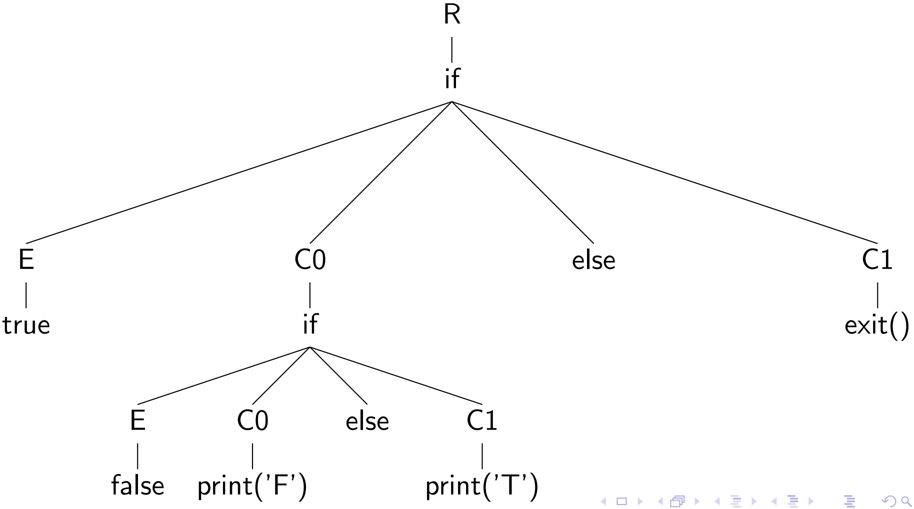
Nested IFs: using two phase annotation

- Consider for instance the input and its AST.
- Input = **if (true)** if (false) print('F') else print('T') **else exit()**



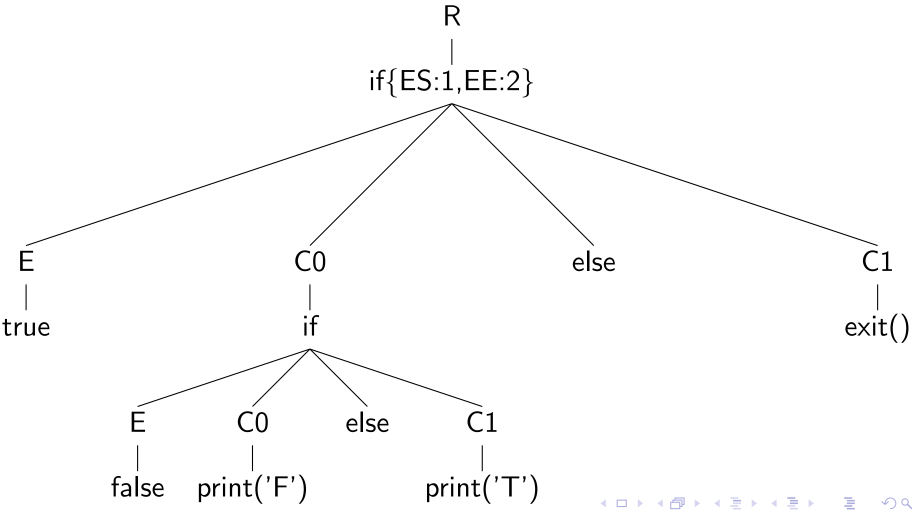
Nested IFs: using two phase annotation

- Through a preorder visit of the tree annotate node settings: iElseStartLabel (ES) and iElseEndLabel (EE)



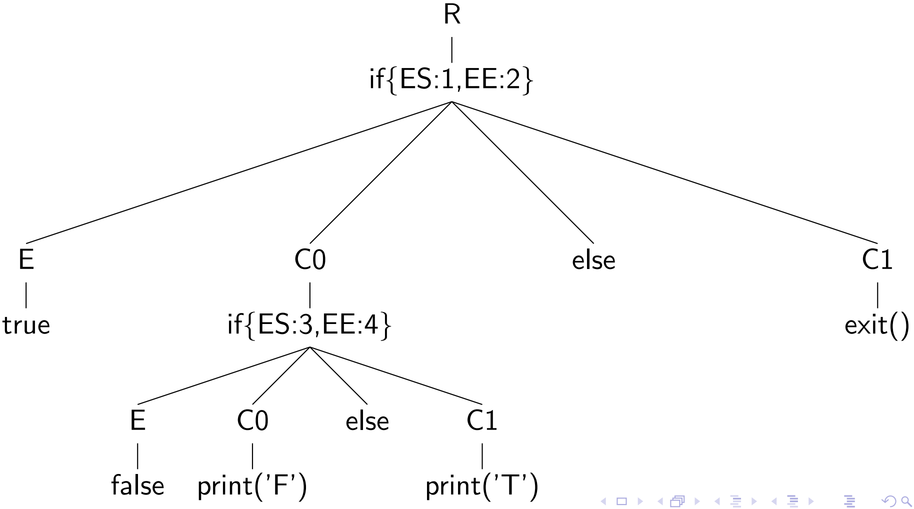
Nested IFs: annotation, first level

- Through a preorder visit of the tree annotate node settings: iElseStartLabel (ES) and iElseEndLabel (EE)



Nested IFs: annotation, second level

- Through a preorder visit of the tree annotate nodes, setting: iElseStartLabel (ES) and iElseEndLabel (EE)



Algorithm 2: IfTreeToPostfixAnnotation

Input: Tree root ρ.

Output: Postfix notation of tree.

```
begin
    Let Node be ρ
    if Node.Operation is "if" then
        print code evaluating Node.E
        print(Node.iElseStartLabel); print("jump_if_false") //M1
        IfTreeToPostfixAnnotation(Node.C0)
        print(Node.iElseEndLabel); print('jump') // M2
        print(Node.iElseStartLabel); printf(':') // M2
        IfTreeToPostfixAnnotation(Node.C1)
        print(iElseEndLabel);printf(':') //M3
    else
        Visit the rest of the tree and print possible commands
    /* Actions annotated by M1, M2, M3 are fragments of code
     initially appearing in the SDT producing the code.*/
```

Nested IFs: generating the code by the algorithm

```
true test
1 jump_if_false
false test
3 jump_if_false
print('F')
4 jump
3:
print('T')
4:
2 jump
1:
exit()
2:
```

- For conveniency the postfix notation has been written using many lines, read it from left to right from top to bottom as a single line to obtain the real result.

Generating target code: summary

- The unfolding of the actions performed by RunPostfixExpression algorithm allows the production of assembler code.
- Instead of pushing elements σ on stack $\mathcal{S}$ output assembler code to push it onto the CPU's stack.

Let $\mathcal{E}$ be a postfix notation expression, e.g. $\mathcal{E} =$ "0 id 3 * 2 + -".

Algorithm 3: CompilePostfixExpression

Input: $\mathcal{E}$ the postfix expression.

```
begin
    foreach symbol σ in E do
        if ¬IsOperator(σ) then
            print('push σ')
        else
            let α be the arity of operator σ
            for (i=0;i< α;i++) do
                print('pop value to i-th register ρ')
            print the code to compute ρ = σ(ρ_0,...,ρ_{α−1})
            print('push(ρ)')
```

Compiling Postfix Notation to target code

- Let $\mathcal{E} =$ "·0 id 3 * 2 + -".

```
/*Empty*/
```

Compiling Postfix Notation to target code

- Let $\mathcal{E}$ = "0·id 3 * 2 + -".

```
push 0
```

Compiling Postfix Notation to target code

- Let $\mathcal{E}$ = "0 id ·3 * 2 + -".

```
push 0
push id
```

Compiling Postfix Notation to target code

- Let $\mathcal{E} =$ "0 id 3 ·* 2 + -".

```
push 0
push id
push 3
```

Compiling Postfix Notation to target code

- Let $\mathcal{E}$ = "0 id 3 * ·2 + -".

```
push 0
push id
push 3
pop eax // *(·,·) operator
pop ebx //has arity 2, pop 2 elements·*/
imul ebx
push eax
```

Compiling Postfix Notation to target code

- Let $\mathcal{E}$ = "0 id 3 * 2 $\cdot$+ -".

```
push 0
push id
push 3
pop eax // *(·,·) operator
pop ebx //has arity 2, pop 2 elements·*/
imul ebx
push eax
push 2
```

- You see the point.

Bibliography

A. V. Aho, R. Sethi and J. D. Ullman
Compilers: Principles, Techniques, and Tools
Addison Wesley, 2007.

D. Brown, J. Levine and T. Mason
Lex & Yacc, 2nd Edition
O'Reilly Media

J. Levine
Flex & Bison
O'Reilly Media