CMPSC 160
Translation of Programming Languages

Lecture 7: LR Parsing + Implementation Optimizations + Abstract Syntax Tree

LR(0), SLR, LR(1)

- Similarity
  - Start with LR(k) items
  - DFA for Handle Recognition
    - L0 as the closure of the production rule with the start non-terminal
    - Iteratively generate other states
  - DFA to Action-Goto Table
  - Parser = stack + Action-Goto Table

- Differences
  - both LR(0) and SLR work on LR(0) items, but SLR uses the follow set to build the Action-Goto table, thus avoiding some conflicts.
  - LR(1) instead work on LR(1) item, a more sophisticated way (look-ahead symbol) to determine when to reduce (where to put reduction in the Action-Goto table)

LR(*k*) items

Examples:

- LR(0) items [$\alpha \rightarrow \beta \bullet \gamma\delta$] (no look-ahead symbol)
- LR(1) items [$\alpha \rightarrow \beta \bullet \gamma\delta$, a] (one token look-ahead)
- LR(2) items [$\alpha \rightarrow \beta \bullet \gamma\delta$, a b] (two token look-ahead) ...

An LR(*k*) item is a pair [*A* , *B*], where

- *A* is a production $\alpha \rightarrow \beta\gamma\delta$ with a • at some position in the *rhs*
- *B* is a look-ahead string of length ≤ *k* (terminal symbols or $)

The • in an item indicates the position of the top of the stack (how much we have already processed from the input)

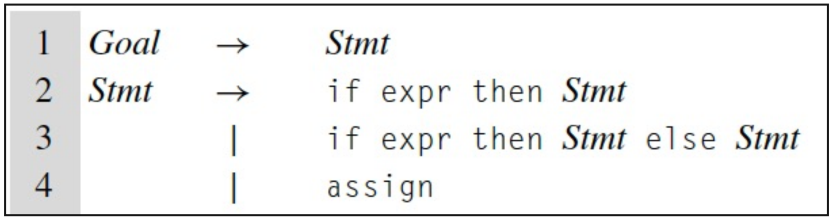
- Examples: [$\alpha \rightarrow \bullet\beta\gamma\delta$, a], [$\alpha \rightarrow \beta\bullet\gamma\delta$, a], [$\alpha \rightarrow \beta\gamma\bullet\delta$, a], [$\alpha \rightarrow \beta\gamma\delta\bullet$, a]

What can go wrong in LR(1) parsing?

Shift/reduce conflict

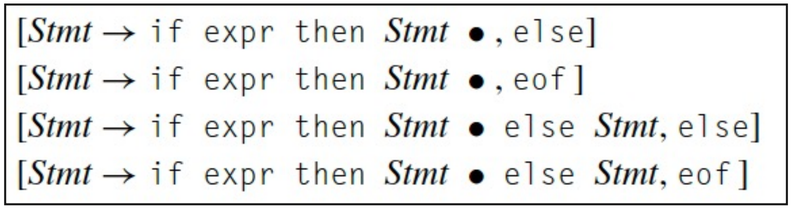
- if a state contains both $[\alpha \rightarrow \beta \bullet a\gamma , b]$ and $[\alpha \rightarrow \beta \bullet , a]$.
- First item generates “shift”, second generates “reduce”

- Example: dangling else problem?



1	*Goal*	→	*Stmt*
2	*Stmt*	→	`if expr then` *Stmt*
3		|	`if expr then` *Stmt* `else` *Stmt*
4		|	`assign`

One state



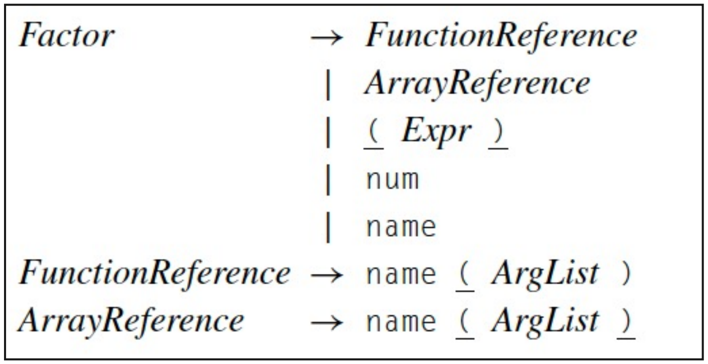
[*Stmt* → `if expr then` *Stmt* •, `else`]
[*Stmt* → `if expr then` *Stmt* •, `eof`]
[*Stmt* → `if expr then` *Stmt* • `else` *Stmt*, `else`]
[*Stmt* → `if expr then` *Stmt* • `else` *Stmt*, `eof`]

  - Modify the grammar to eliminate it
  - Shifting will often resolve it correctly
  - Try LR(k), k>1

What can go wrong in LR(1) parsing?

Reduce/reduce conflict

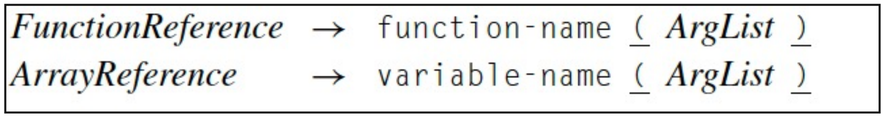
- if a state contains both $[\alpha \rightarrow \beta \bullet , a]$ and $[\gamma \rightarrow \beta \bullet , a]$
- Each generates "reduce", but with a different production
- Example: function call vs. array reference



Factor → *FunctionReference*
| *ArrayReference*
| (*Expr*)
| `num`
| `name`
FunctionReference → `name` (*ArgList*)
ArrayReference → `name` (*ArgList*)

Since the last two productions have identical right-hand sides, this grammar is ambiguous, which creates a reduce-reduce conflict in an LR(1) table builder.

  - Modify the grammar to eliminate it
  - Try LR(k), k>1

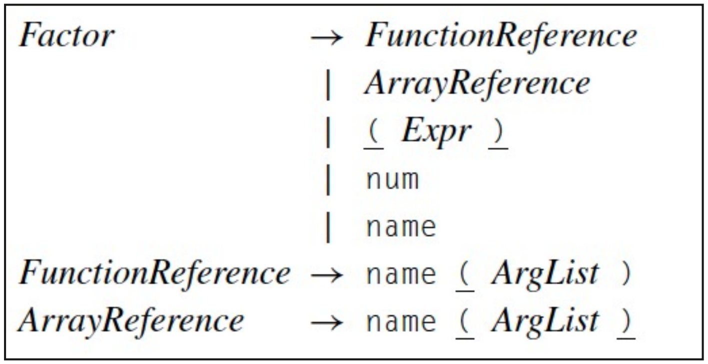


FunctionReference → `function-name` (*ArgList*)
ArrayReference → `variable-name` (*ArgList*)

What can go wrong in LR(1) parsing?

Reduce/reduce conflict

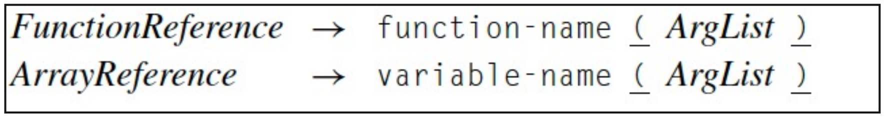
- if a state contains both [α → β • , a] and [γ → β • , a]
- Each generates "reduce", but with a different production
- Example: function call vs. array reference



Factor	→	*FunctionReference*
	|	*ArrayReference*
	|	(*Expr*)
	|	num
	|	name
FunctionReference	→	name (*ArgList*)
ArrayReference	→	name (*ArgList*)

Since the last two productions have identical right-hand sides, this grammar is ambiguous, which creates a reduce-reduce conflict in an LR(1) table builder.

- Modify the grammar to eliminate it
- Try LR(k), k>1

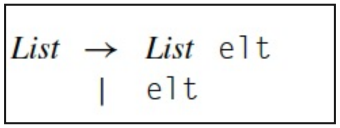


FunctionReference	→	function-name (*ArgList*)
ArrayReference	→	variable-name (*ArgList*)

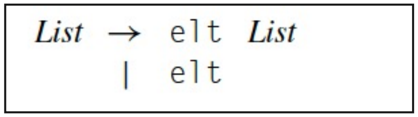
Left Recursion vs Right Recursion

- For LL(k): only right recursion
- For LR(k) parsers: Both are acceptable

- Which is more better?
  - Left recursion is not only compiler writer friendly, but turns out to be more memory efficient.



```
List → List elt
     | elt
```



```
List → elt List
     | elt
```

  - Input: elt elt elt elt elt
  - What is the size of the stack during parsing?
    - The right-recursive grammar requires more stack space; its maximum stack depth is bounded only by the length of the list. In contrast, the maximum stack depth with the left-recursive grammar depends on the grammar rather than the input stream.

How to check whether a grammar is LR(k)?

Build a LR(k) parser for it!
Check whether there are any conflicts.

Parser

- Input?
    - Input program as a series of tokens

- Output?
    - accept or error
    - accept: parse tree
    - error: error message

Parse Trees

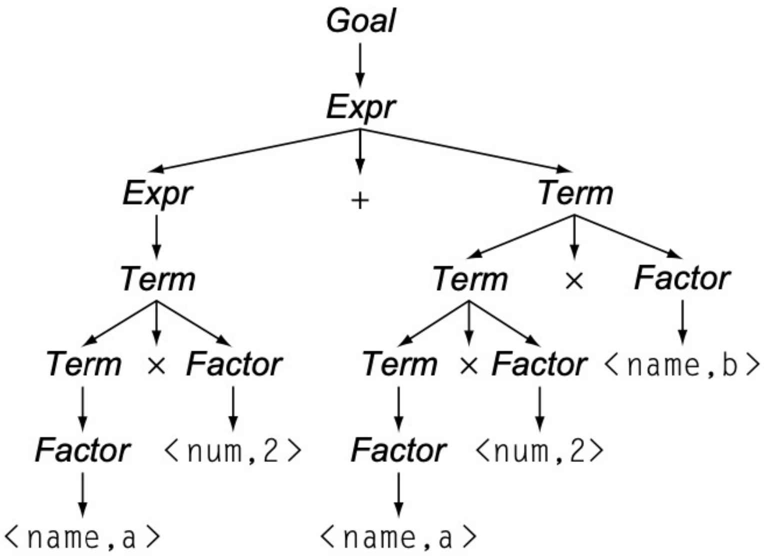
- The *parse tree* is a graphical representation for the derivation, or parse, that corresponds to the input program.

Goal → *Expr*

Expr → *Expr* + *Term*
    | *Expr* - *Term*
    | *Term*

Term → *Term* × *Factor*
    | *Term* ÷ *Factor*
    | *Factor*

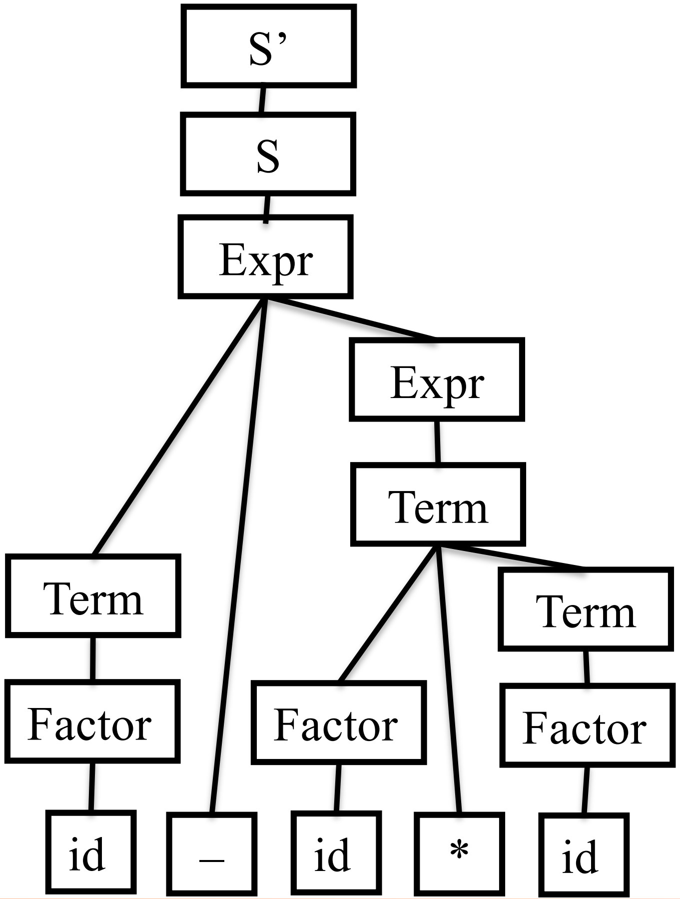
Factor → (*Expr*)
    | `num`
    | `name`



Parse tree generation for x-z*y

0 $S' \rightarrow S$
1 $S \rightarrow Expr$
2 $Expr \rightarrow Term - Expr$
3 $Expr \rightarrow Term$
4 $Term \rightarrow Factor * Term$
5 $Term \rightarrow Factor$
6 $Factor \rightarrow \text{id}$

Stack	Input	Action
0	id – id * id $	S4
0 id 4	– id * id $	**R6**, G3
0 F 3	– id * id $	**R5**, G2
0 T 2	– id * id $	S5
0 T 2 – 5	id * id$	S4
0 T 2 – 5 id 4	* id$	**R6**, G3
0 T 2 – 5 F 3	* id$	S6
0 T 2 – 5 F 3 * 6	id$	S4
0 T 2 – 5 F 3 * 6 id 4	$	**R6**,G3
0 T 2 – 5 F 3 * 6 F 3	$	**R5**,G8
0 T 2 – 5 F 3 * 6 T 8	$	**R4**,G2
0 T 2 – 5 T 2	$	**R3**,G7
0 T 2 – 5 E 7	$	**R2**,G1
0 E 1	$	**R1**
0 S 9	$	
Accept	$	



Error recovery

- **Panic-mode recovery:** On discovering an error, discard input symbols one at a time until one synchronizing token is found
    - For example delimiters such as “;” or “}” can be used as synchronizing tokens
- **Phrase-level recovery:** On discovering an error make local corrections to the input
    - For example replace “,” with “;”
- **Error-productions:** If we have a good idea about what type of errors occur, we can augment the grammar with error productions and generate appropriate error messages when an error production is used
- **Global correction:** Given an incorrect input string try to find a correct string which will require minimum changes to the input string
    - In general, too costly

Advanced Optimizations for Efficient Parser Implementations

1. Parse tree size
    - A top-down parser performs an expansion for every production in the derivation.
    - A bottom-up parser performs a reduction for every production in the derivation.
    - Tree size → **number of derivations**

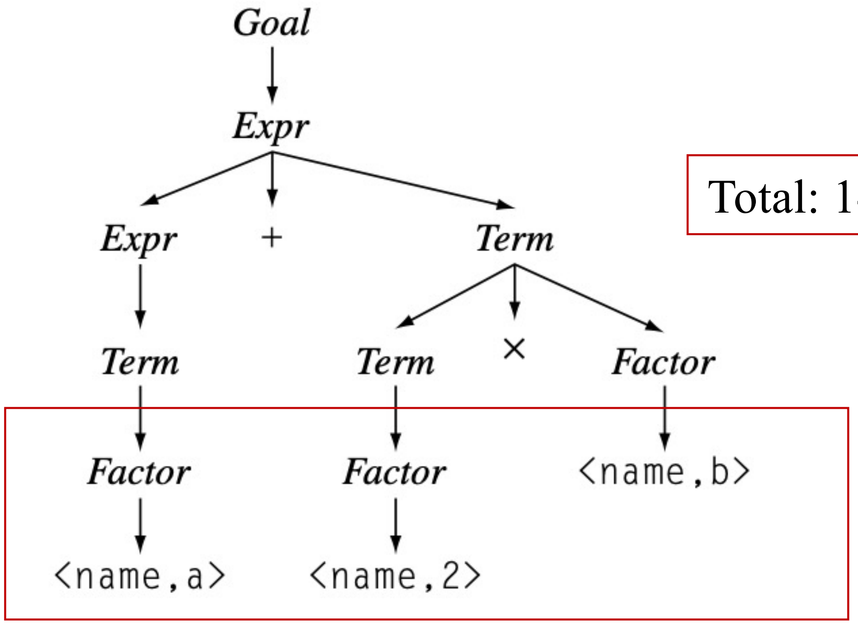
2. Table size
    - **Table lookup efficiency**: Action and Goto tables in LR
    - Reduce table size?

Optimization1: Reduce the Parse Tree size

- **Key insight:** A grammar that produces shorter derivations takes less time to parse.
- **Who**: the compiler writer
- **How:** Examine transformations on the grammar that reduce the length of a derivation to produce a faster parse.
- Apply to both LL and LR parser.

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* x *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	`num`
9		|	`name`

(a) The Classic Expression Grammar



(b) Parse Tree for a + 2 x b

Total: 14 nodes

Any interior node that has only one child is a candidate for optimization

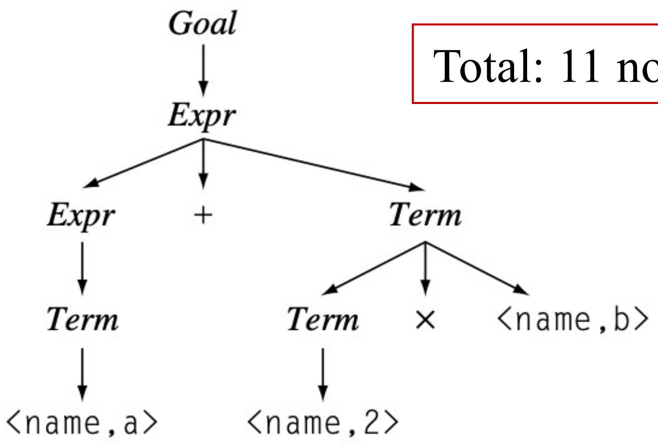
Optimization1: Reduce the Parse Tree size

- We can eliminate at least one layer, the layer of *Factor* nodes, by folding the alternative expansions for *Factor* into *Term*,

```
4   Term   → Term x Factor
5          | Term ÷ Factor
6          | Factor
7   Factor → ( Expr )
8          | num
9          | name
```

```
4   Term → Term x ( Expr )
5        | Term x name
6        | Term x num
7        | Term ÷ ( Expr )
8        | Term ÷ name
9        | Term ÷ num
10       | ( Expr )
11       | name
12       | num
```

(a) New Productions for *Term*



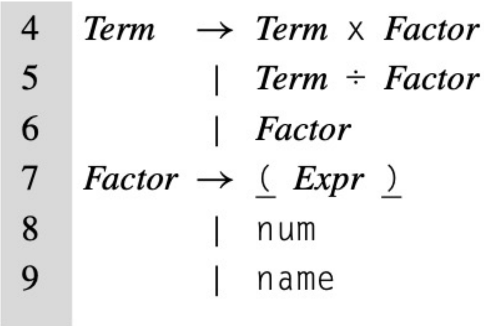
Total: 11 nodes

(b) Parse Tree for a + 2 x b

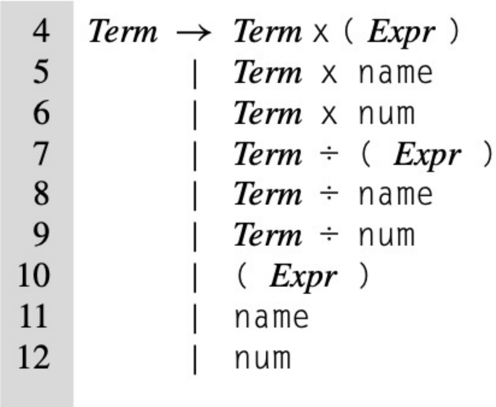
In a top-down recursive-descent parser for an equivalent predictive grammar, it would eliminate 3 of 14 procedure calls.

Optimization1: Reduce the Parse Tree size

- For LR parsing, the improvement is more subtle.
  - We can also reduce the #reductions
    - eliminates three of nine reduce actions, and leaves the five shifts intact (9+5=14 → 9+3=11)
  - but the Action and Goto table size ….
    - Action table: #states * #terminals
    - Goto table: #states * #non-terminals



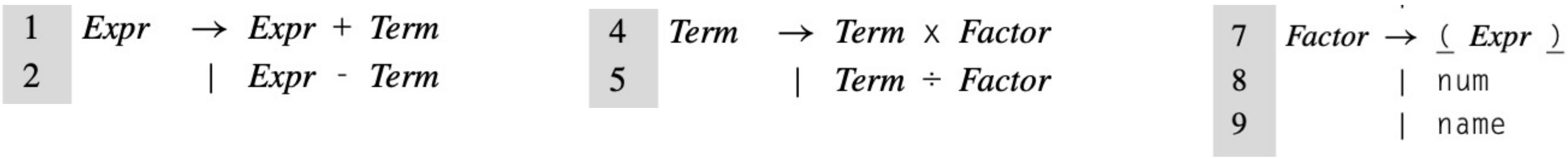
4	*Term*	→	*Term* x *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	`num`
9		|	`name`



4	*Term*	→	*Term* x (*Expr*)
5		|	*Term* x `name`
6		|	*Term* x `num`
7		|	*Term* ÷ (*Expr*)
8		|	*Term* ÷ `name`
9		|	*Term* ÷ `num`
10		|	(*Expr*)
11		|	`name`
12		|	`num`

- In our example, eliminating *Factor* removes one column from the Goto table, but the extra productions for *Term* increase the size of states from 32 to 46 sets. Thus, the tables have one fewer column, but an extra 14 rows.
- The resulting parser performs fewer reductions (and runs faster), but has larger tables.

Other Optimizations



1 $Expr \rightarrow Expr + Term$
2 $\quad | \; Expr - Term$

4 $Term \rightarrow Term \times Factor$
5 $\quad | \; Term \div Factor$

7 $Factor \rightarrow \underline{(} \; Expr \; \underline{)}$
8 $\quad | \; \texttt{num}$
9 $\quad | \; \texttt{name}$

- Add/subtraction (multiplication/division, number/name) makes no difference from the perspective of parsing.

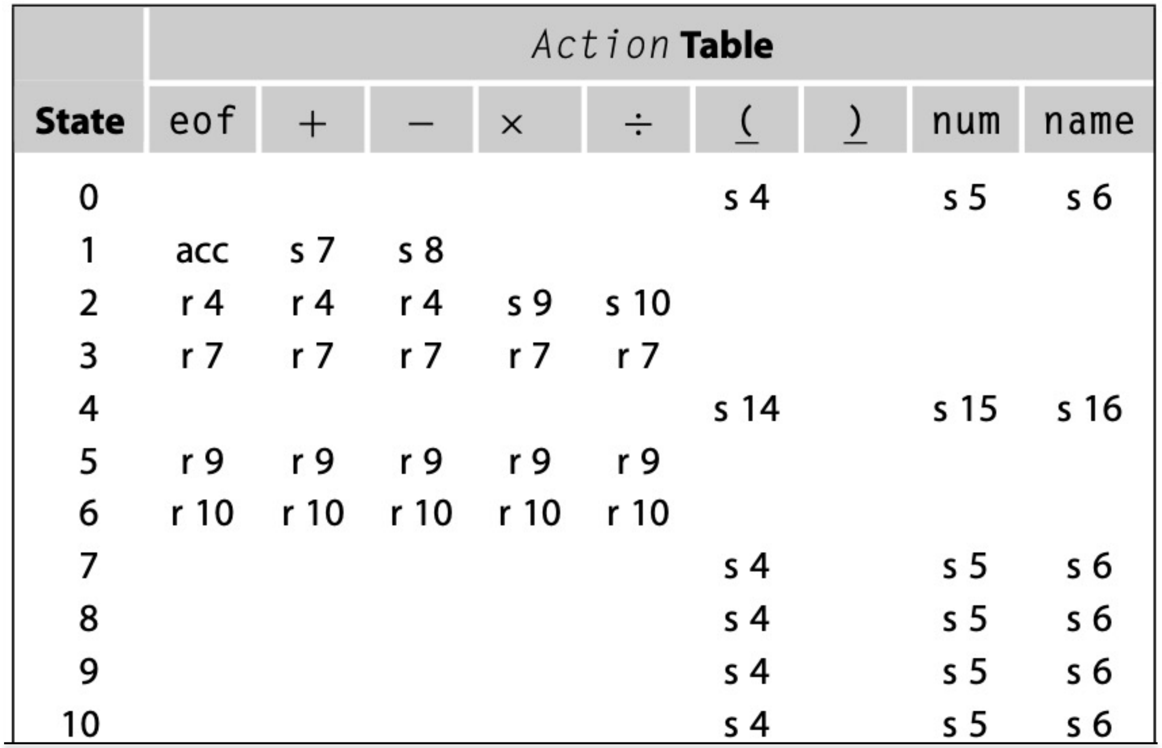
- LL parsing: the code for both nontrivial expansions of *Expr and Term above* is identical.

- Solution: the compiler writer could assign both + and - to the same syntactic category, and the code could be merged together and only use the lexeme to differentiate between the two when needed.

- Similar analysis for LR: the table sizes could also be reduced.

Optimization2: Reducing the Size of LR Tables

- ***Combining Rows or Columns***
  - If the table generator can find two rows, or two columns, that are identical, it can combine them.



	Action Table								
State	eof	+	–	×	÷	(	)	num	name
0						s 4		s 5	s 6
1	acc	s 7	s 8						
2	r 4	r 4	r 4	s 9	s 10				
3	r 7	r 7	r 7	r 7	r 7				
4						s 14		s 15	s 16
5	r 9	r 9	r 9	r 9	r 9				
6	r 10	r 10	r 10	r 10	r 10				
7						s 4		s 5	s 6
8						s 4		s 5	s 6
9						s 4		s 5	s 6
10						s 4		s 5	s 6

E.g., Row 0, 7-10.

  - The table generator can combine identical columns in the analogous way.

Optimization2: Reducing the Size of LR Tables

- Combining rows and columns produces a direct reduction in table size. If this space reduction adds an extra indirection to every table access, the cost of those memory operations must trade off directly against the savings in memory.

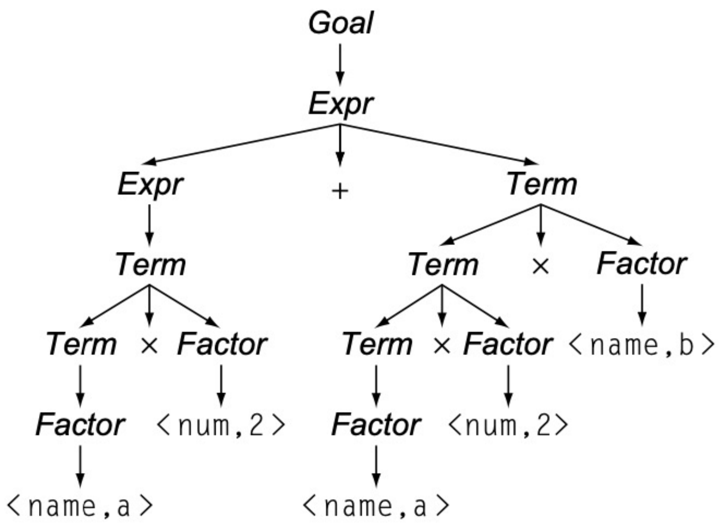
- The table generator could also use other techniques to represent sparse matrices—again, the implementor must consider the tradeoff of memory size against any increase in access costs.

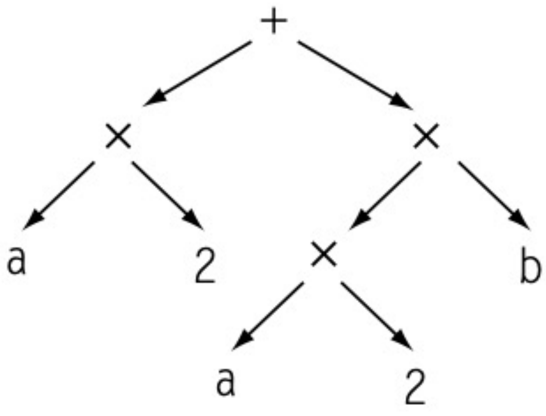
Parser Implementation

- Optimizing the grammar cannot change the parser’s asymptotic behavior. Still, reducing the constants in heavily used portions of the grammar, such as the expression grammar, can make enough difference to justify the effort.

Parse Trees V.S. Abstract Syntax Trees

- Since the compiler must allocate memory for each node and each edge, and it must traverse all those nodes and edges during compilation, it is worth considering ways to **shrink this parse tree**.
  - the key is to abstract away those nodes that serve no real purpose in the rest of the compiler.





- This approach leads to a simplified version of the parse tree, called an abstract syntax tree.
  - The precedence and meaning of the expression remain, but extraneous nodes have disappeared.

Abstract Syntax Trees (ASTs)

What exactly is an Abstract Syntax Tree (AST) in practice?

- It is basically a data structure (a.k.a., intermediate representation) that is used to represent the input program for facilitating program analysis and compiler optimization.
  - All the information that we need to analyze the program and to translate the program to the target language is available in the AST
  - The syntactic details about the input are not kept in the AST since we do not need them after the parsing is over

- AST is a tree shaped data structure corresponding to the recursive nature of the abstract syntax of the program

Intermediate Representations (IRs): Overview

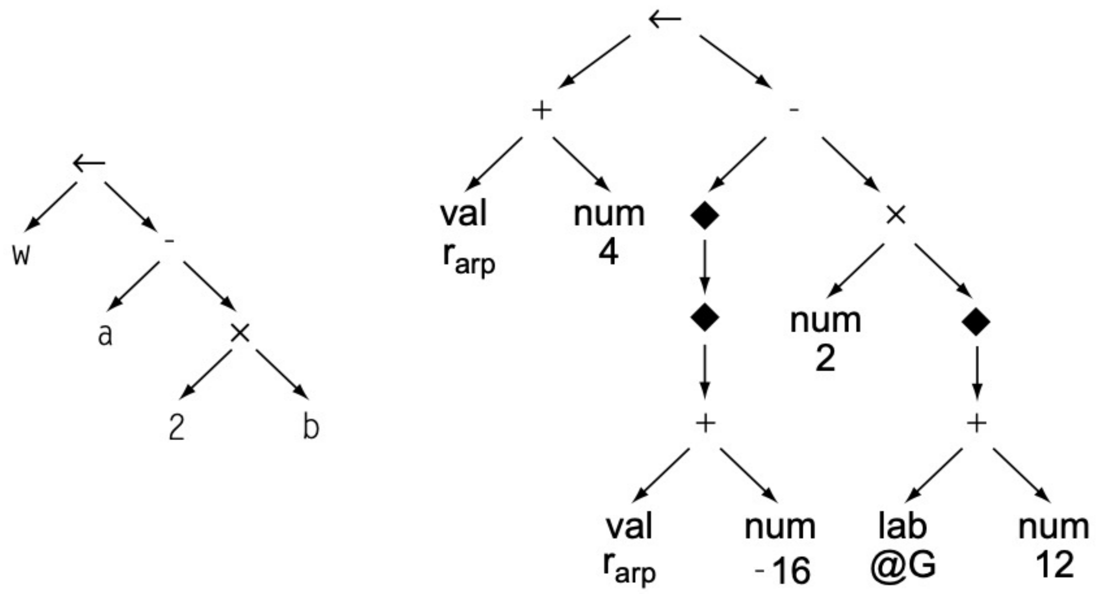
- There is more than one **data structure** to represent code as it is being generated, analyzed, and optimized.

- Graphical IRs
  - **Abstract Syntax Trees (AST)**
  - Directed Acyclic Graphs (DAG)
  - Control Flow Graphs (CFG)
- Linear IRs
  - Static Single Assignment Form (SSA)
  - Stack Machine Code
  - Three Address Code

The best data structure to use depends on the **specific optimization** we want to conduct.

ASTs with Different Levels of Abstraction

- Many compilers and interpreters use ASTs, but the level of abstraction that those systems need varies widely.



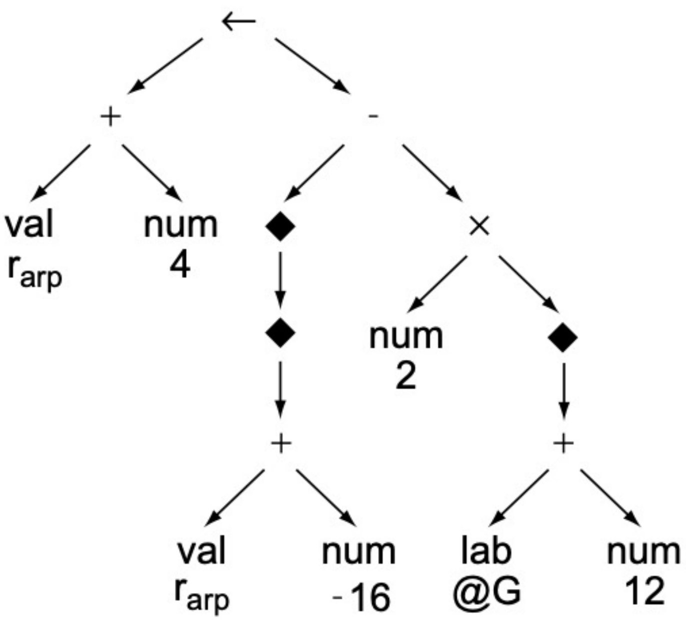
(a) Source-Level AST (b) Low-Level AST

- The source-level tree lacks much of the detail needed to translate the statement into assembly code.
- A low-level tree with four new node types can make that detail explicit.

ASTs with Different Levels of Abstraction

Compiler can, in general, only optimize details that are exposed in the IR.

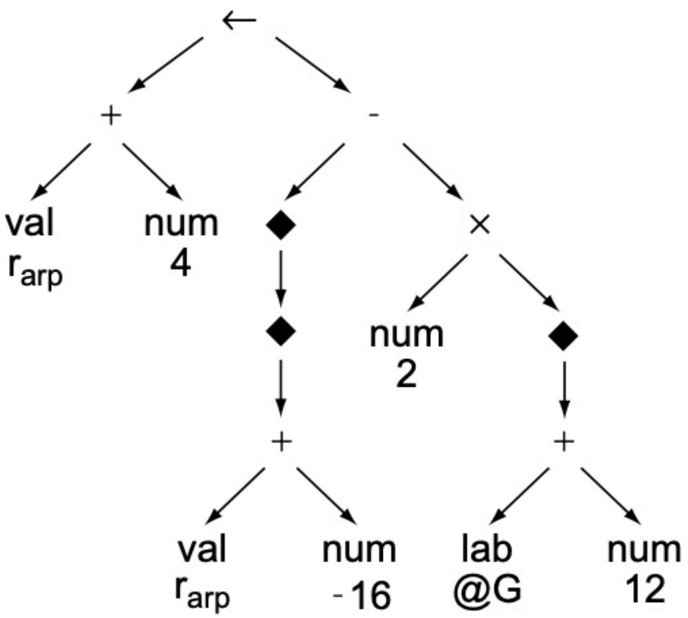
- Properties that are implicit in the IR are hard to change, in part because the compiler would need to translate implicit facts in different, instance-specific ways



(b) Low-Level AST

- A val node represents a value already in a register.
- A num node represents a known constant.
- A lab node represents an assembly-level label, typically a relocatable symbol.
- ◆ is an operator that dereferences a value; it treats the value as a memory address and returns the contents of the memory at that address.

ASTs with Different Levels of Abstraction



(b) Low-Level AST

The low-level tree reveals the address calculations for the three variables.

- w is stored at offset 4 from the pointer in rarp, which holds the pointer to the data area for the current procedure.
- The double dereference of a shows that it is a call-by-reference formal parameter accessed through a pointer stored 16 bytes before rarp.
- Finally, b is stored at offset 12 after the label @G, where A lab node represents an assembly-level label, typically a relocatable symbol.

Compiler can, in general, only optimize details that are exposed in the IR.

- Properties that are implicit in the IR are hard to change, in part because the compiler would need to translate implicit facts in different, instance-specific ways