LECTURE 8

Parsing

OVERVIEW

Last lecture, we talked about the difference between LL and LR grammars. In this lecture, we will be focusing on LL grammars. Even though an LL grammar tends to be more difficult to understand, an LL parser is simpler to write.

There are two types of LL parsers which we will cover: Recursive Descent Parsers and Table-Driven Top-Down Parsers.

We talked about how to write a Recursive Descent Parser for an LL grammar. Today we'll give an example use of one.

For the remainder of the lecture, we will introduce Table-Driven parsing.

RECURSIVE DESCENT PARSER

Let’s review the grammar we used last time.

We created subroutines for each of the non-terminal symbols. Each subroutine is responsible for identifying valid tokens that can be encountered at that point, and either matching them or calling another subroutine.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

RECURSIVE DESCENT PARSER

Let’s see our parser in action. Let’s say we write the following code:

(1 + 2) * 3

Our scanner breaks it up into the following tokens:

‘(‘ int ‘+’ int ‘)’ ‘*’ int ‘$’

Remember, we have an end-of-input signifier (such as EOF).

RECURSIVE DESCENT PARSER

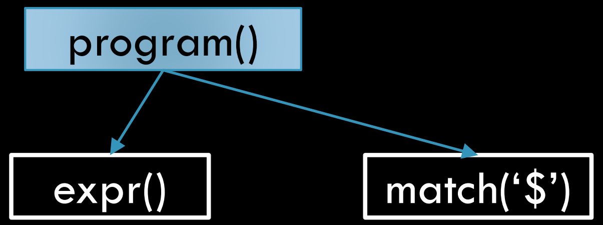
‘(‘ int ‘+’ int ‘)’ ‘*’ int ‘$’

⬆

We will keep track of our tokens above, and the *call graph* is at the top-right for reference.

Now let’s look at the subroutine we created.

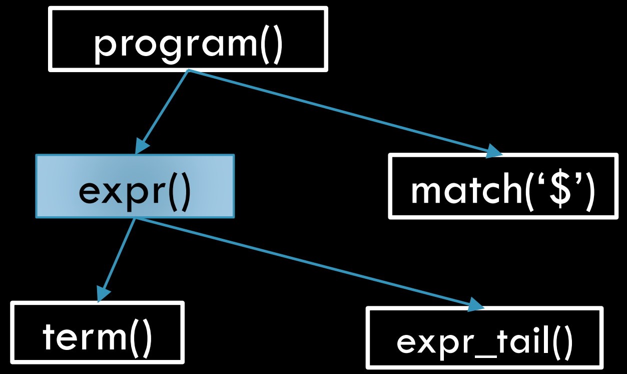
RECURSIVE DESCENT PARSER



‘(‘ int ‘+’ int ‘)’ ‘*’ int ‘$’

```
procedure program
    case input of:
        '(', int: expr() match('$')
        else error
```

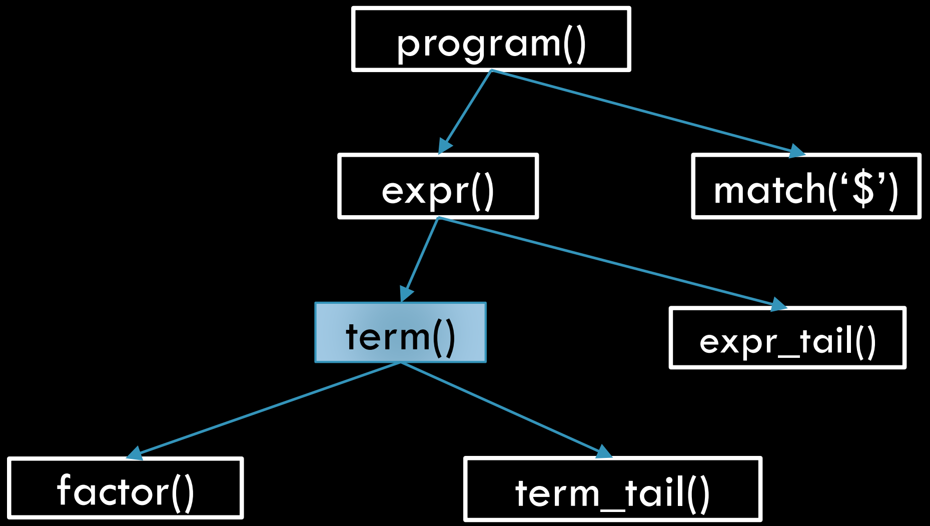
RECURSIVE DESCENT PARSER



'('   int   '+'   int   ')'   '*'   int   '$'

```
procedure expr
    case input of:
        '(', int: term() expr_tail()
        else error
```

RECURSIVE DESCENT PARSER

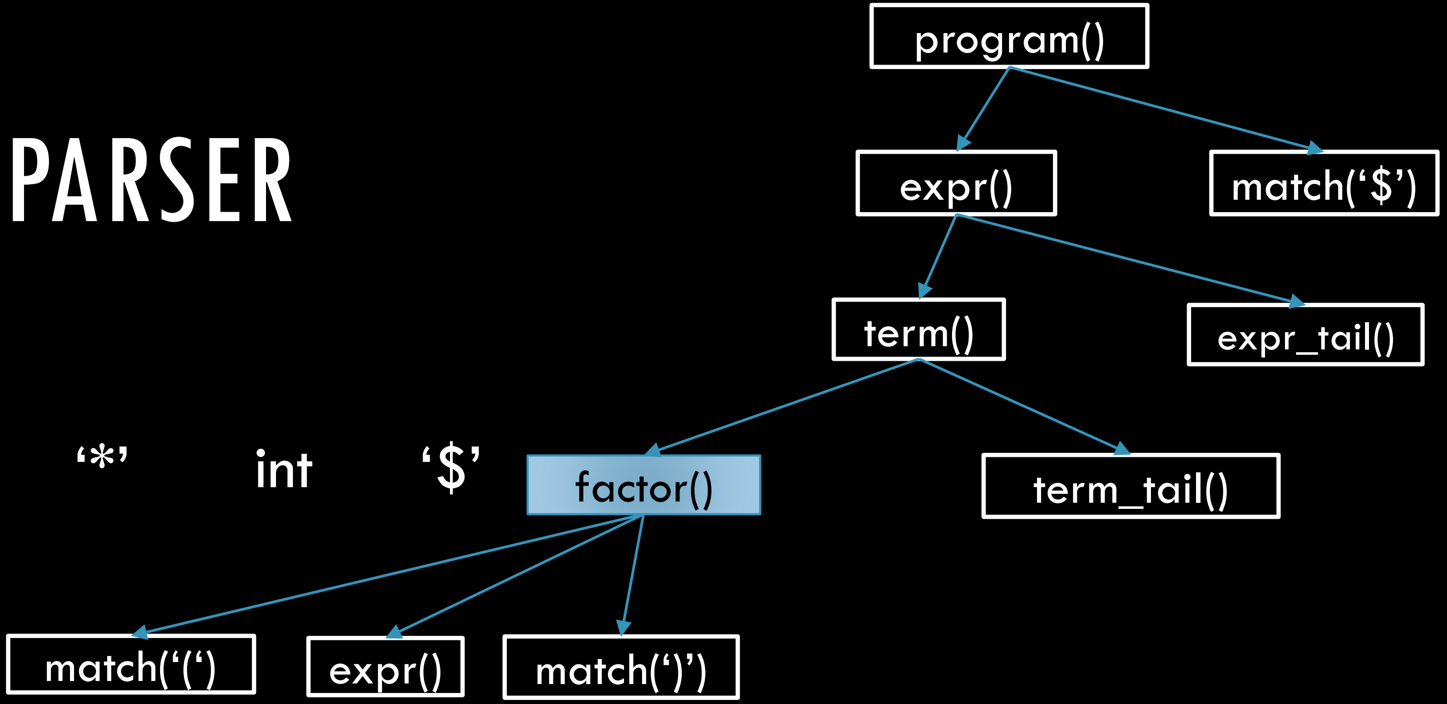


'('   int   '+'   int   ')'   '*'   int   '$'

```
procedure term
    case input of:
        '(', int: factor() term_tail()
        else error
```

RECURSIVE DESCENT PARSER

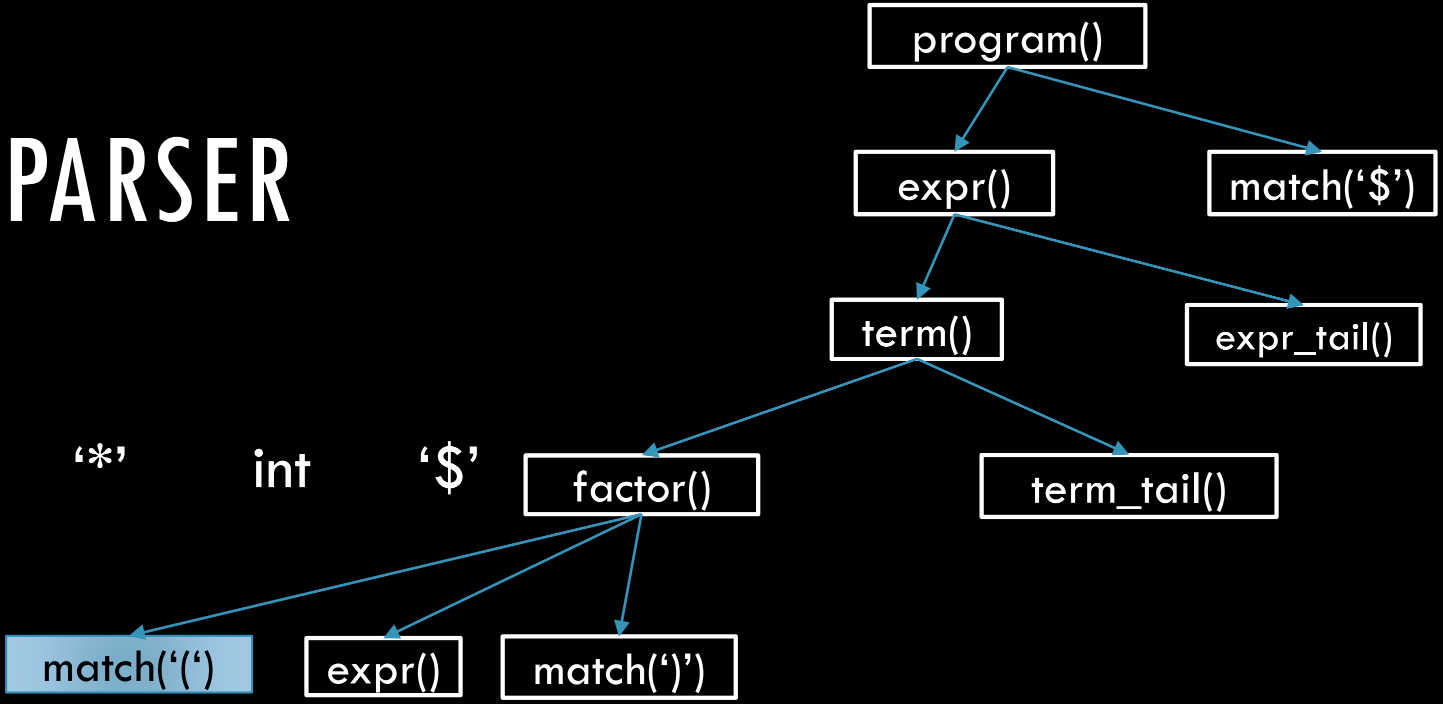
‘(‘ int ‘+’ int ‘)’ ‘*’ int ‘$’



```
procedure factor
    case input of:
        '(': match('(') expr() match(')')
        int: match(int)
        else error
```

RECURSIVE DESCENT PARSER

‘(‘ int ‘+’ int ‘)’ ‘*’ int ‘$’

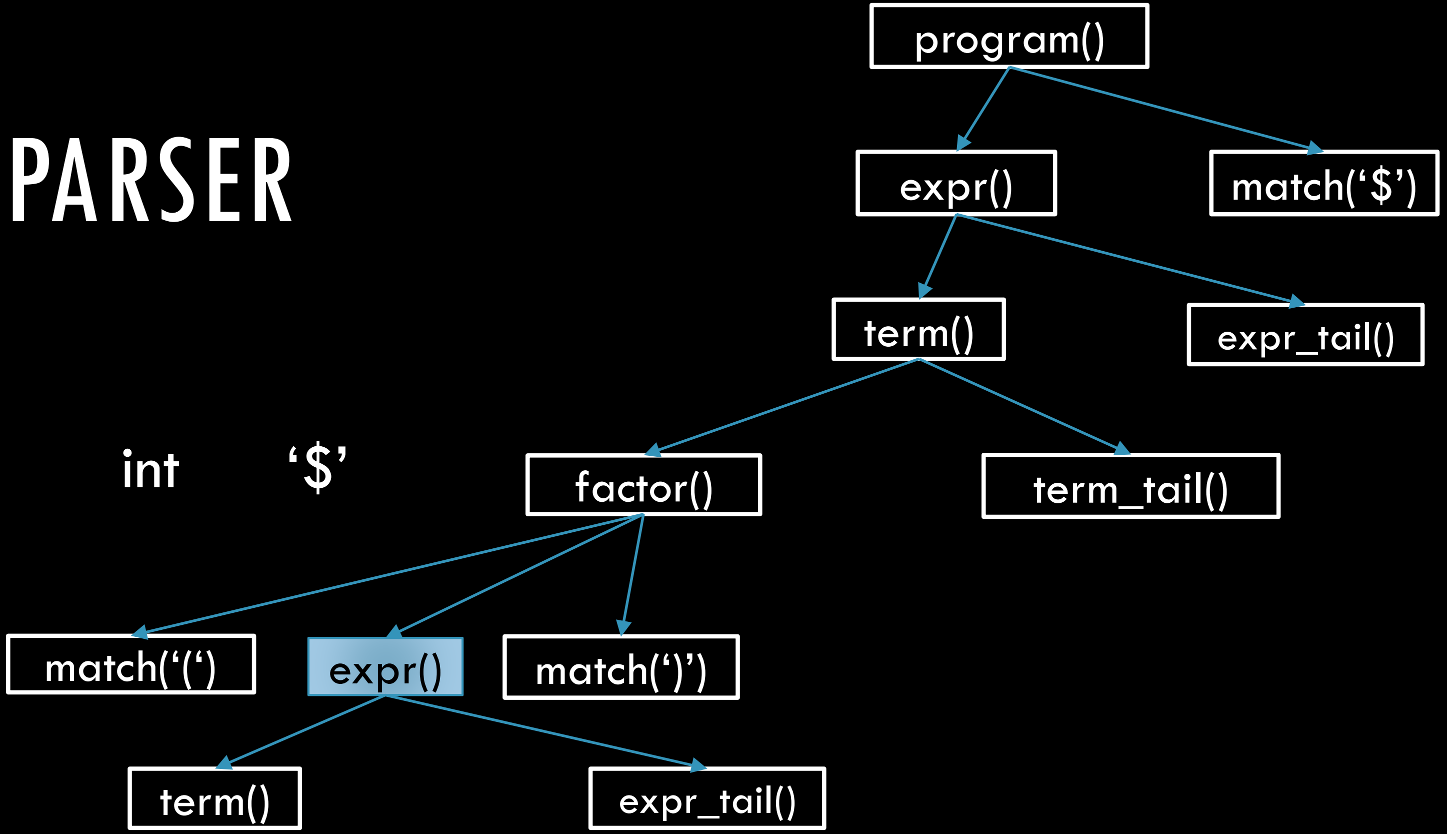


```
procedure factor
    case input of:
        '(': match('(') expr() match(')')
        int: match(int)
        else error
```

RECURSIVE DESCENT PARSER

int ‘+’ int ‘)’ ‘*’ int ‘$’

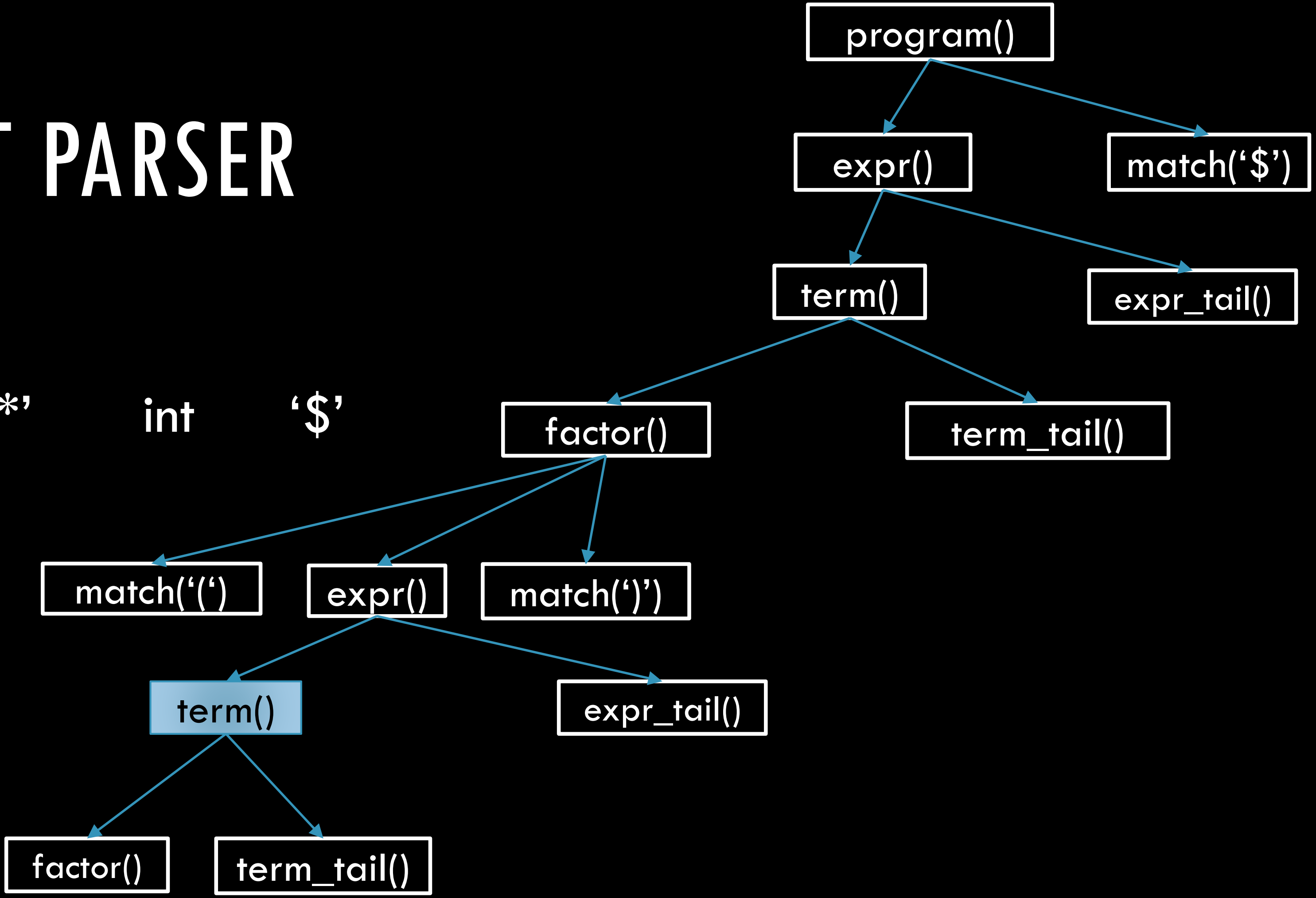
```
procedure expr
    case input of:
        '(', int: term() expr_tail()
        else error
```



RECURSIVE DESCENT PARSER

int ‘+’ int ‘)’ ‘*’ int ‘$’

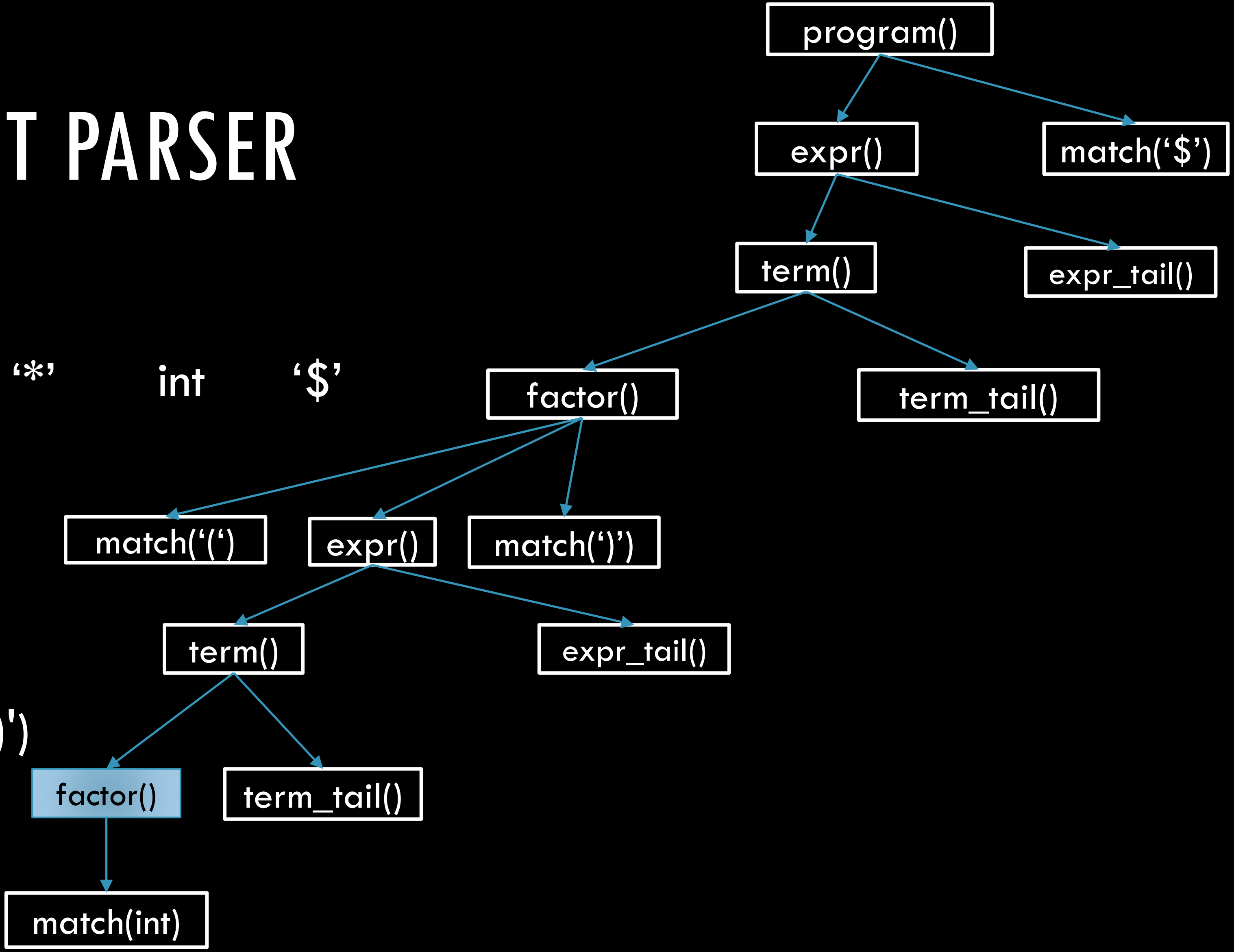
```
procedure term
    case input of:
        '(', int: factor() term_tail()
        else error
```



RECURSIVE DESCENT PARSER

int '+' int ')' '*' int '$'

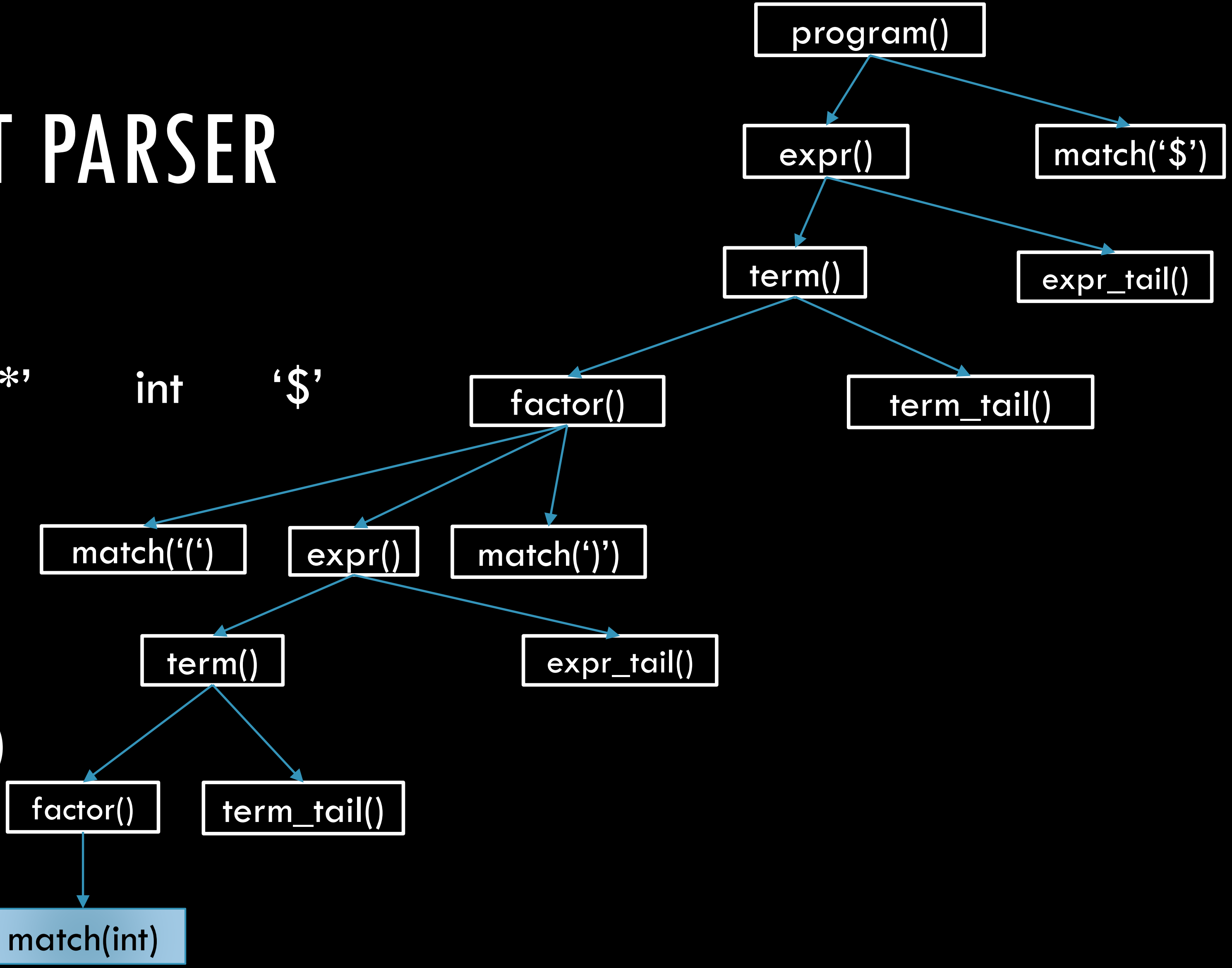
```
procedure factor
    case input of:
        '(': match('(') expr() match(')')
        int: match(int)
        else error
```



RECURSIVE DESCENT PARSER

int ‘+’ int ‘)’ ‘*’ int ‘$’

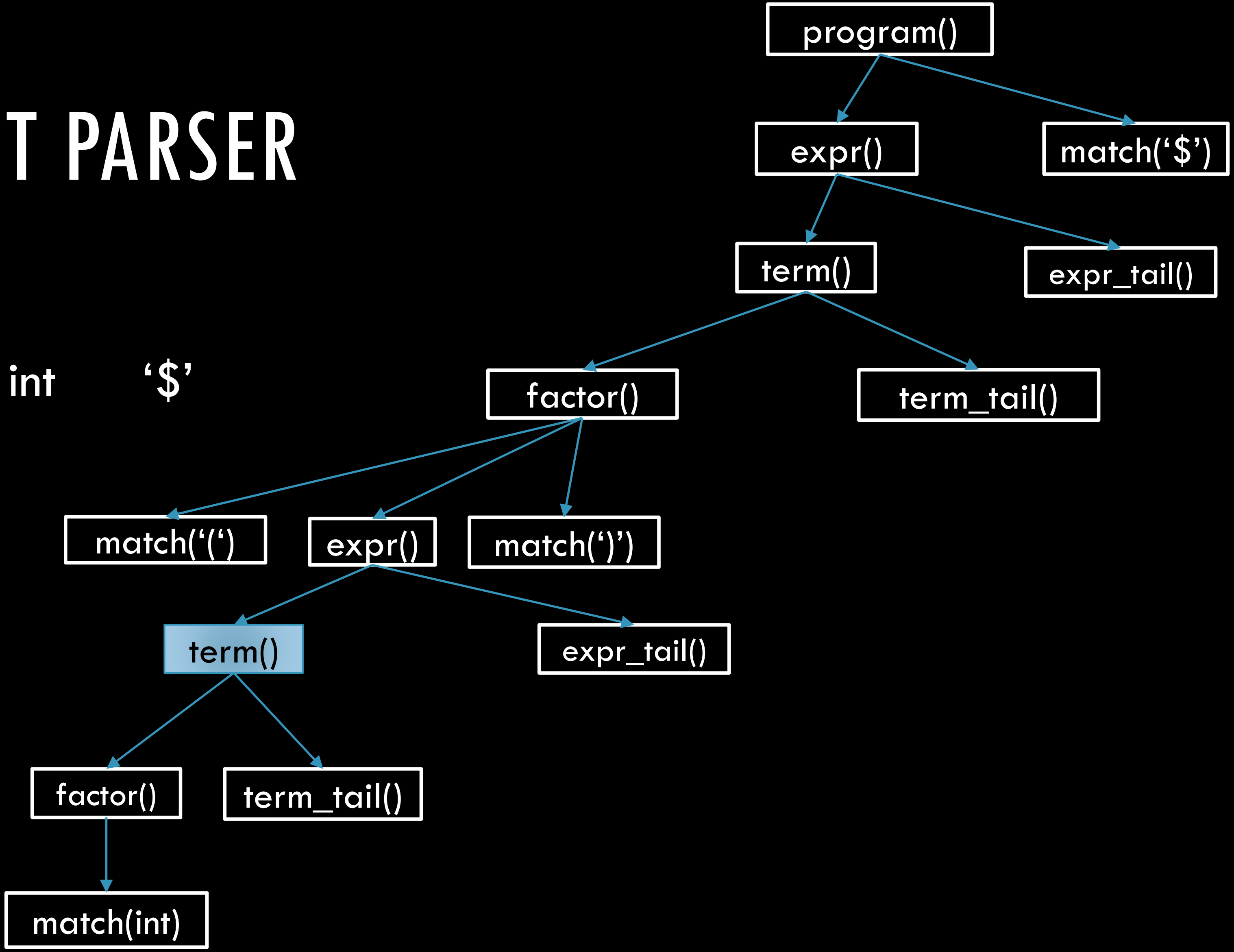
```
procedure factor
    case input of:
        '(': match('(') expr() match(')')
        int: match(int)
        else error
```



RECURSIVE DESCENT PARSER

‘+’ int ‘)’ ‘*’ int ‘$’

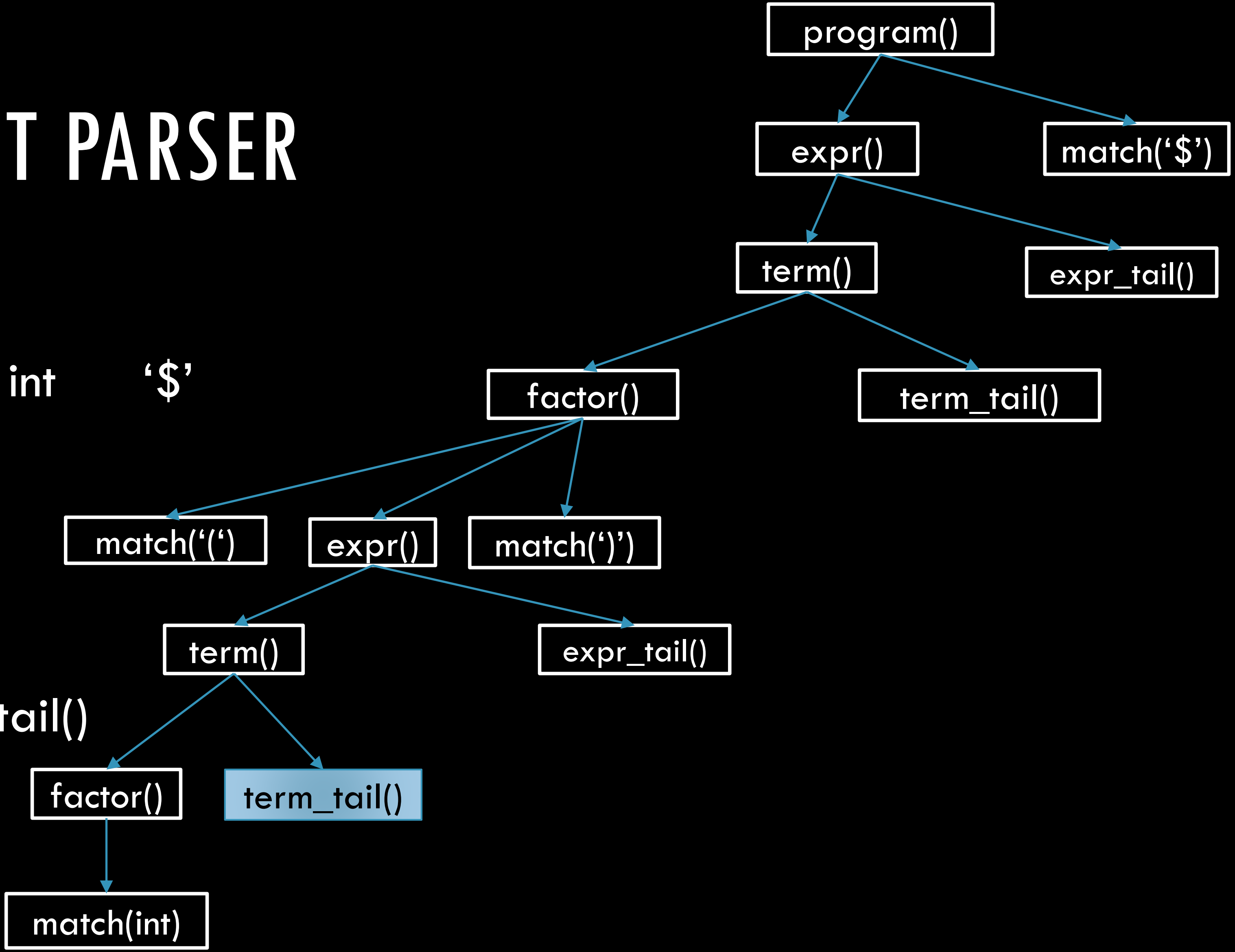
```
procedure term
    case input of:
        '(', int: factor() term_tail()
        else error
```



RECURSIVE DESCENT PARSER

‘+’ int ‘)’ ‘*’ int ‘$’

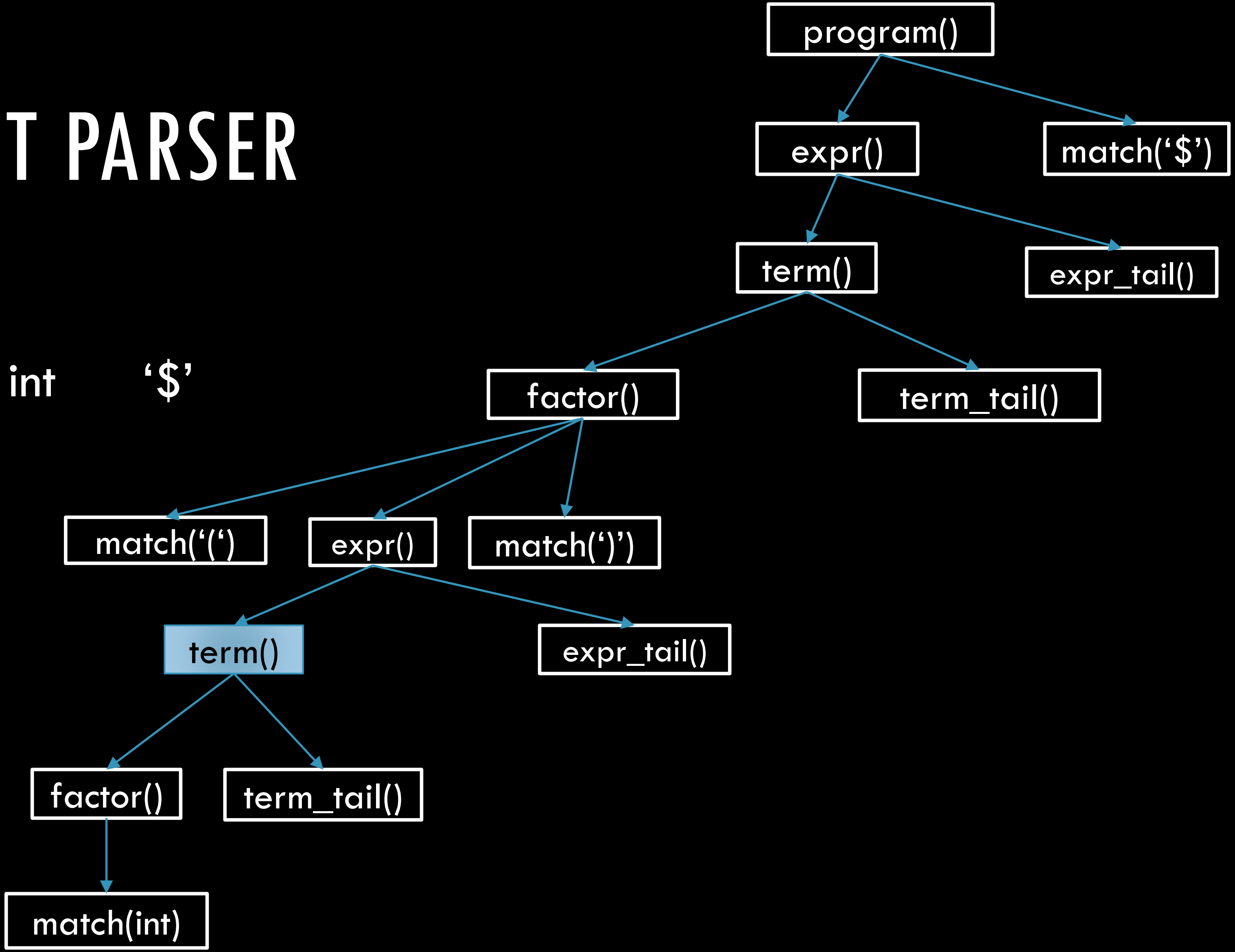
```
procedure term_tail
    case input of:
        '*': match('*') factor() term_tail()
        ‘)’, '+', '$': skip
        else error
```



RECURSIVE DESCENT PARSER

‘+’ int ‘)’ ‘*’ int ‘$’

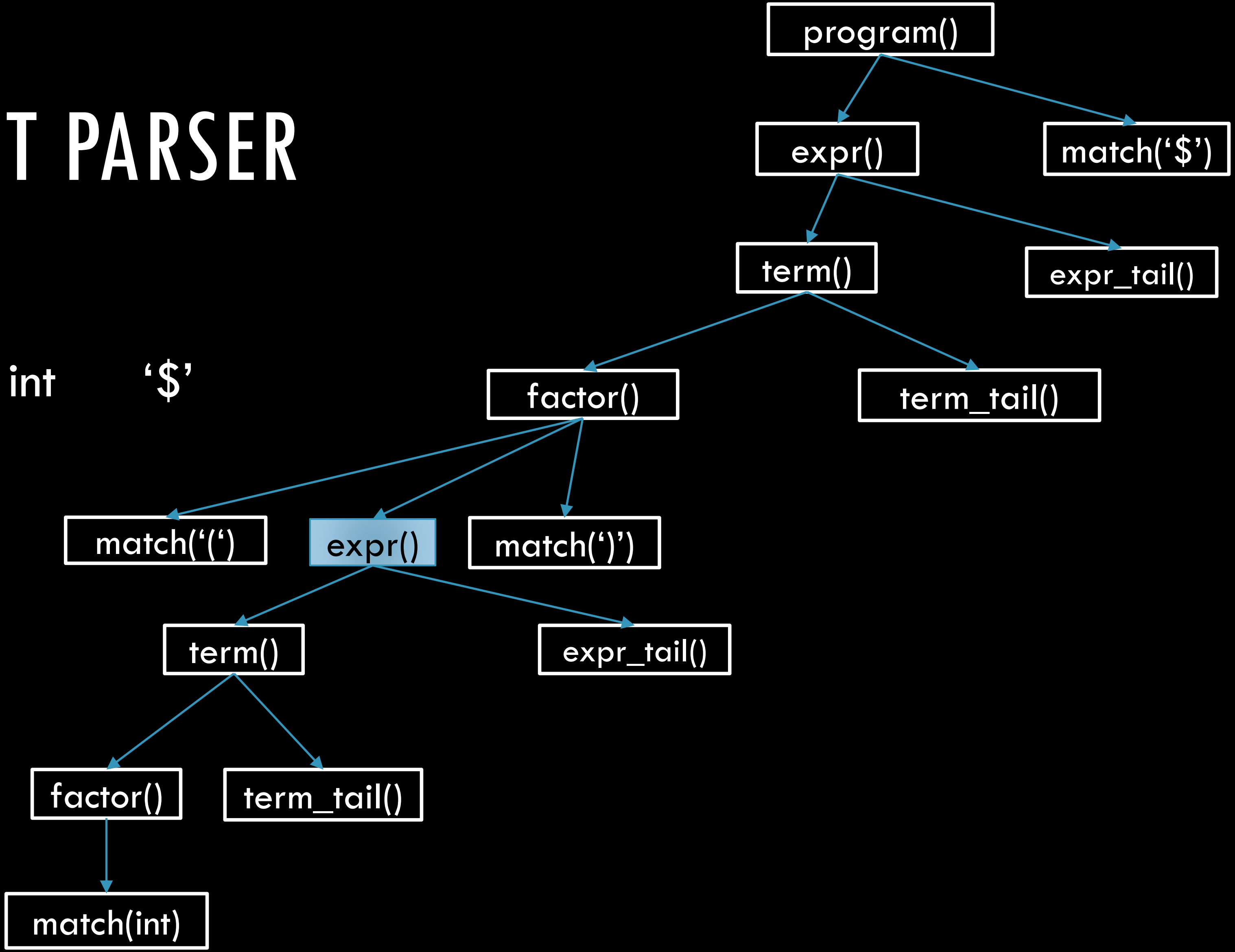
```
procedure term
    case input of:
        '(', int: factor() term_tail()
        else error
```



RECURSIVE DESCENT PARSER

‘+’ int ‘)’ ‘*’ int ‘$’

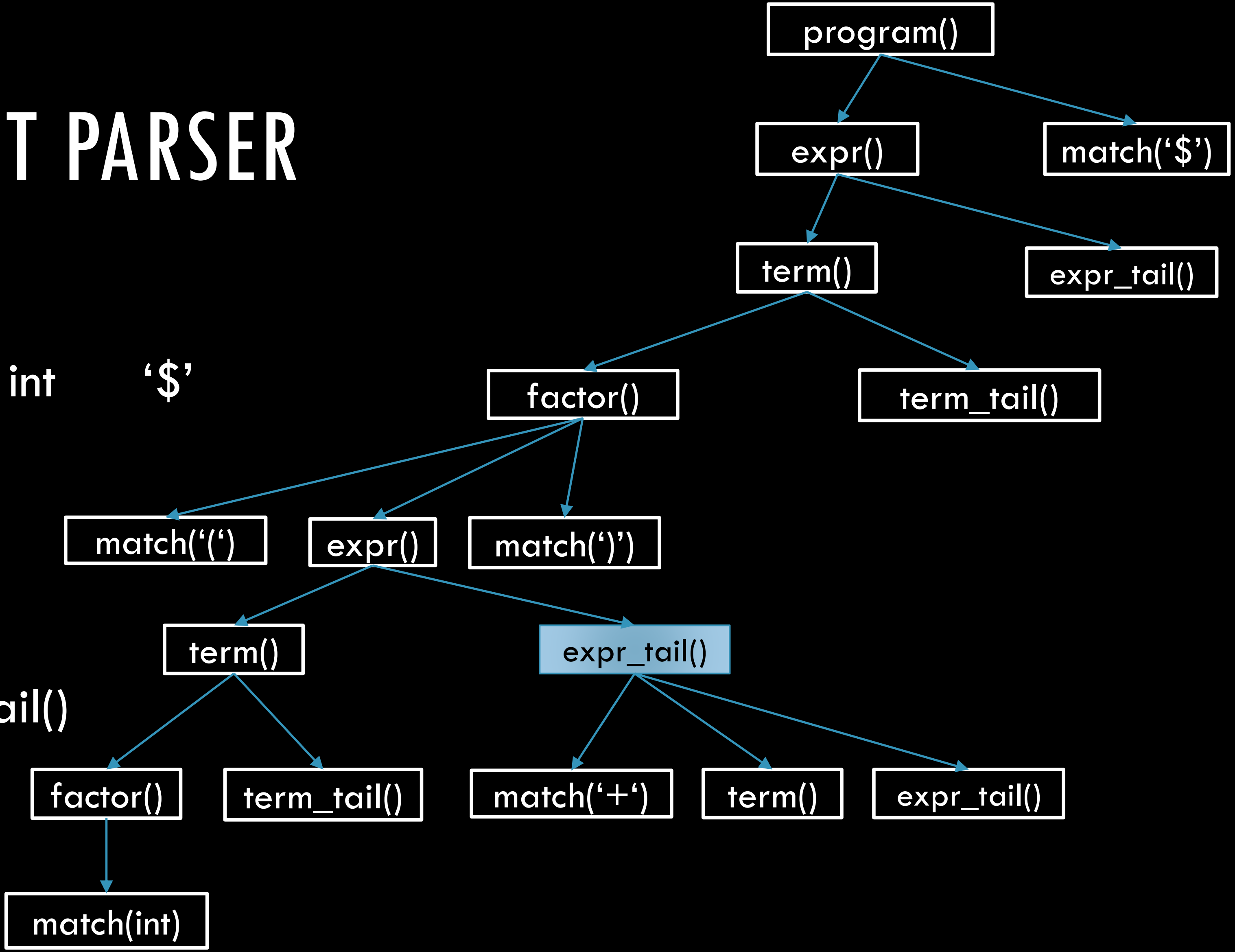
```
procedure expr
    case input of:
        '(', int: term() expr_tail()
        else error
```



RECURSIVE DESCENT PARSER

‘+’ int ‘)’ ‘*’ int ‘$’

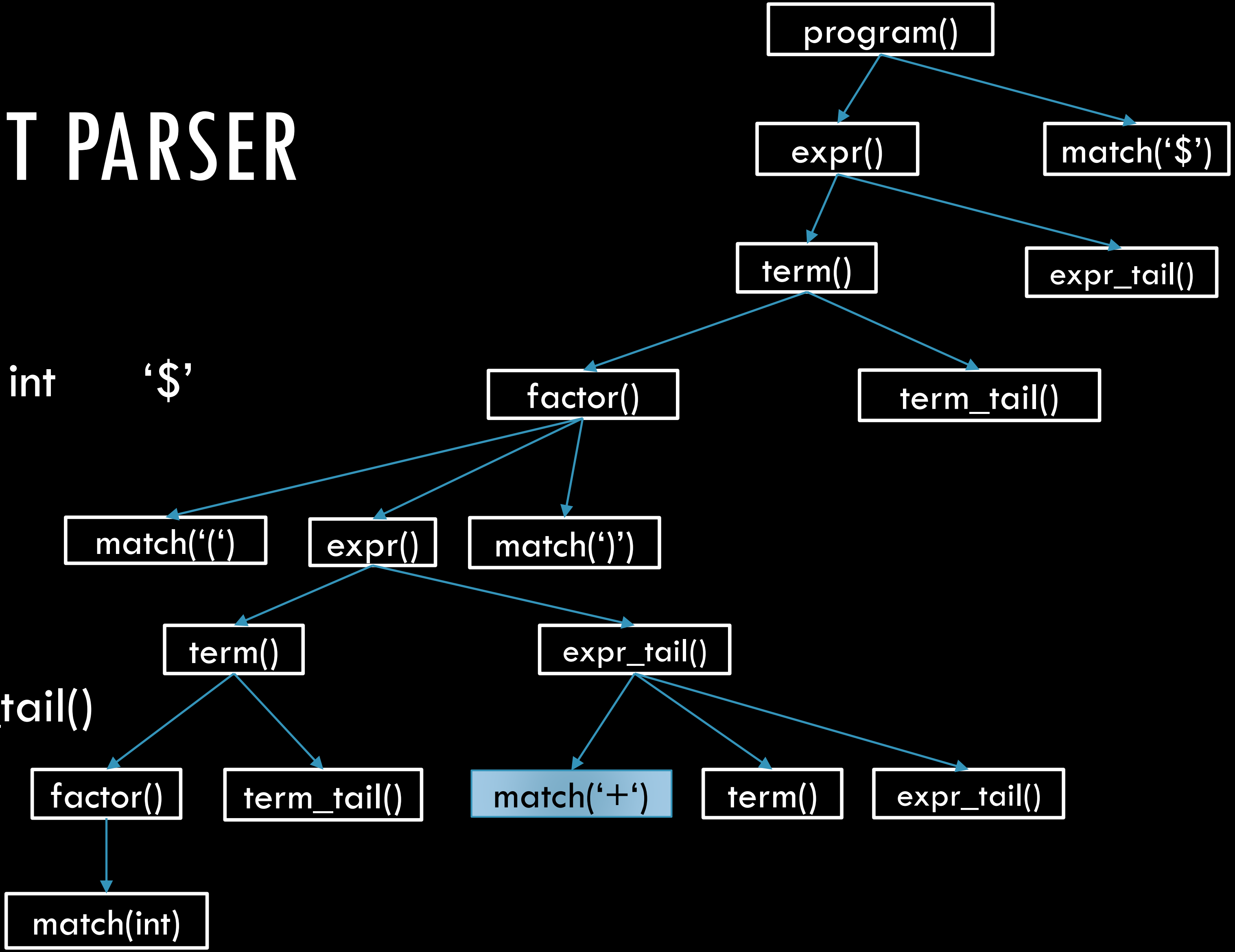
```
procedure expr_tail
    case input of:
        '+': match('+') term() expr_tail()
        ‘)’, '$': skip
        else error
```



RECURSIVE DESCENT PARSER

‘+’ int ‘)’ ‘*’ int ‘$’

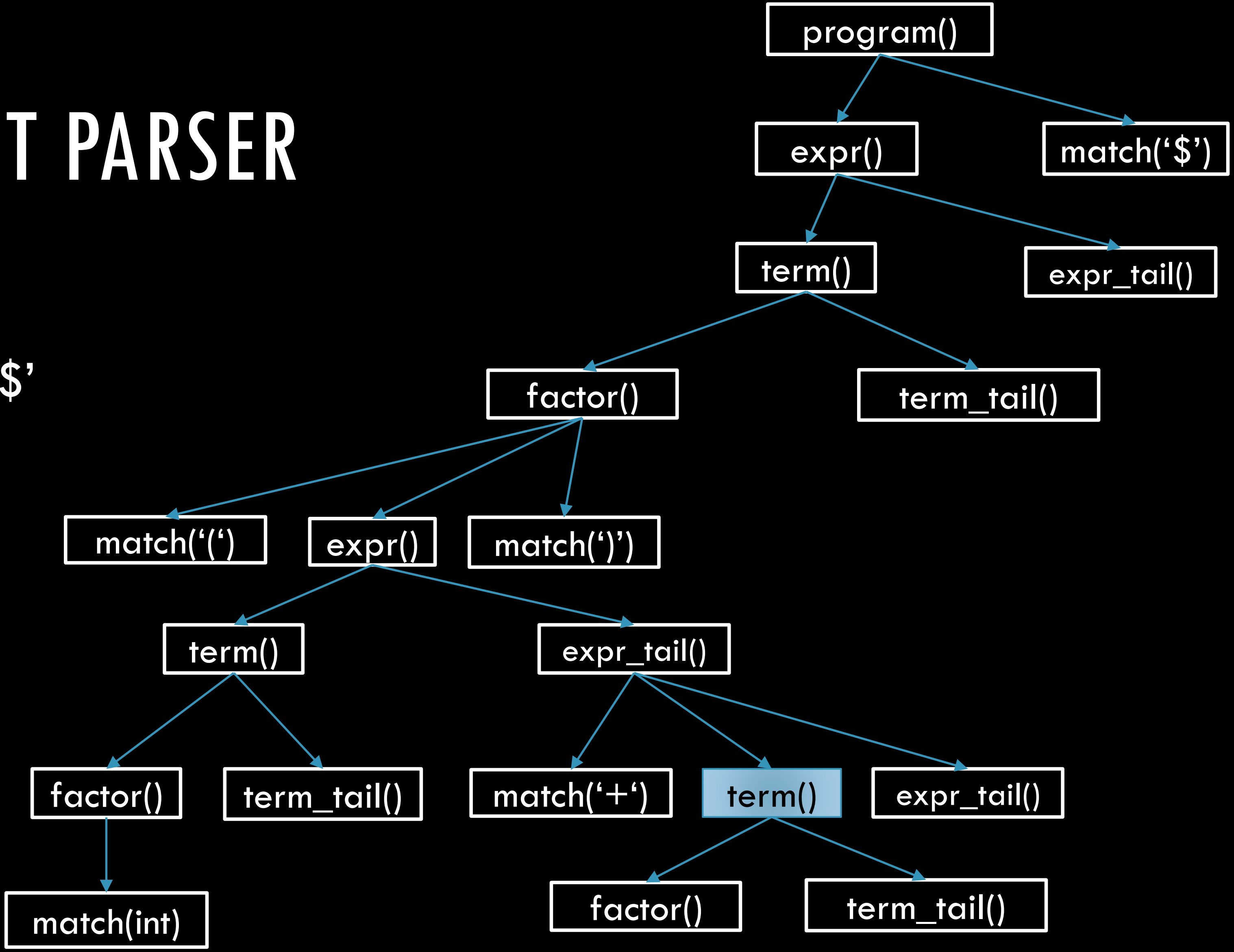
```
procedure expr_tail
    case input of:
        '+': match('+') term() expr_tail()
        ‘)’, '$': skip
        else error
```



RECURSIVE DESCENT PARSER

int ‘)’ ‘*’ int ‘$’

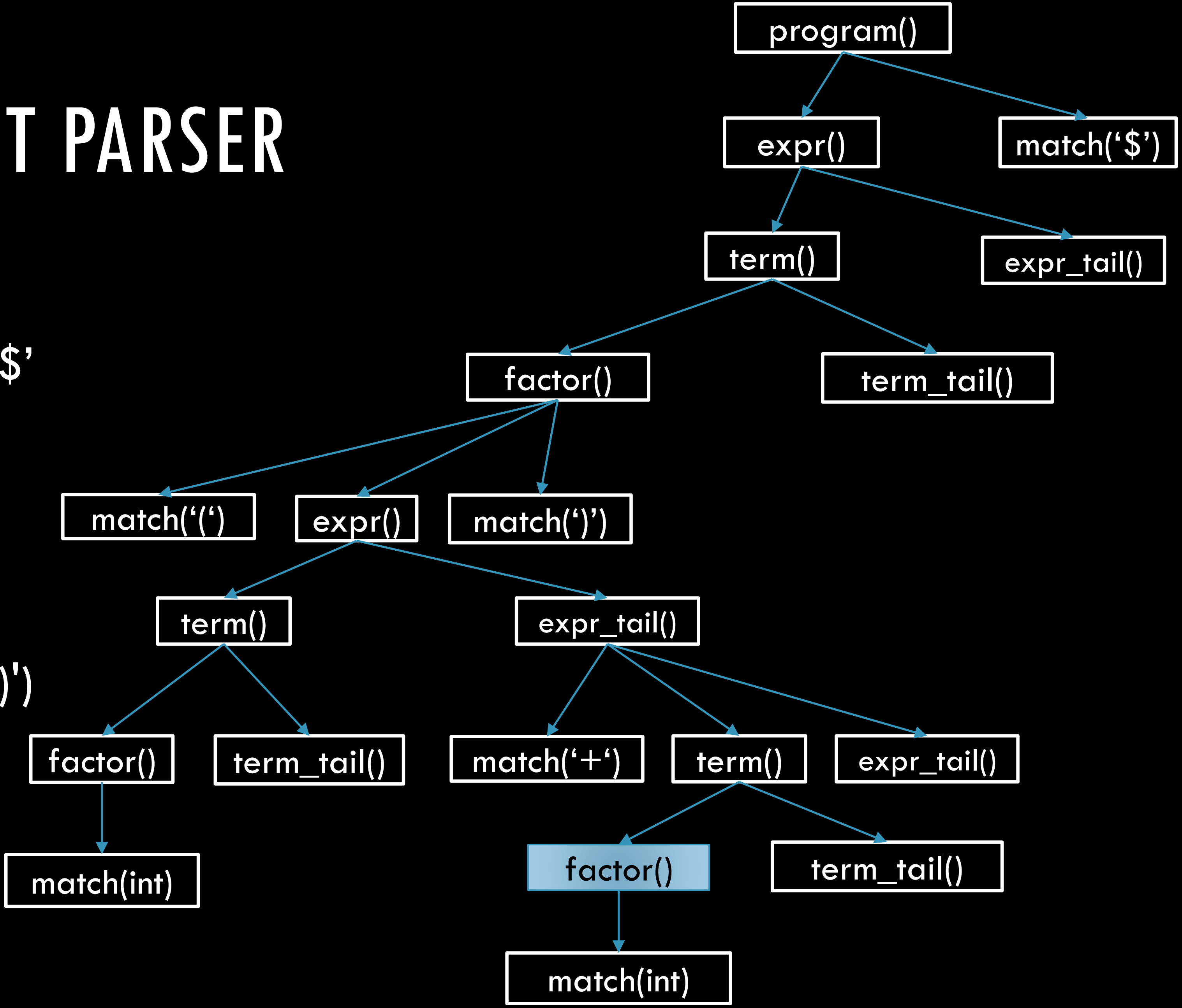
```
procedure term
    case input of:
        '(', int: factor() term_tail()
        else error
```



RECURSIVE DESCENT PARSER

int ')' '*' int '$'

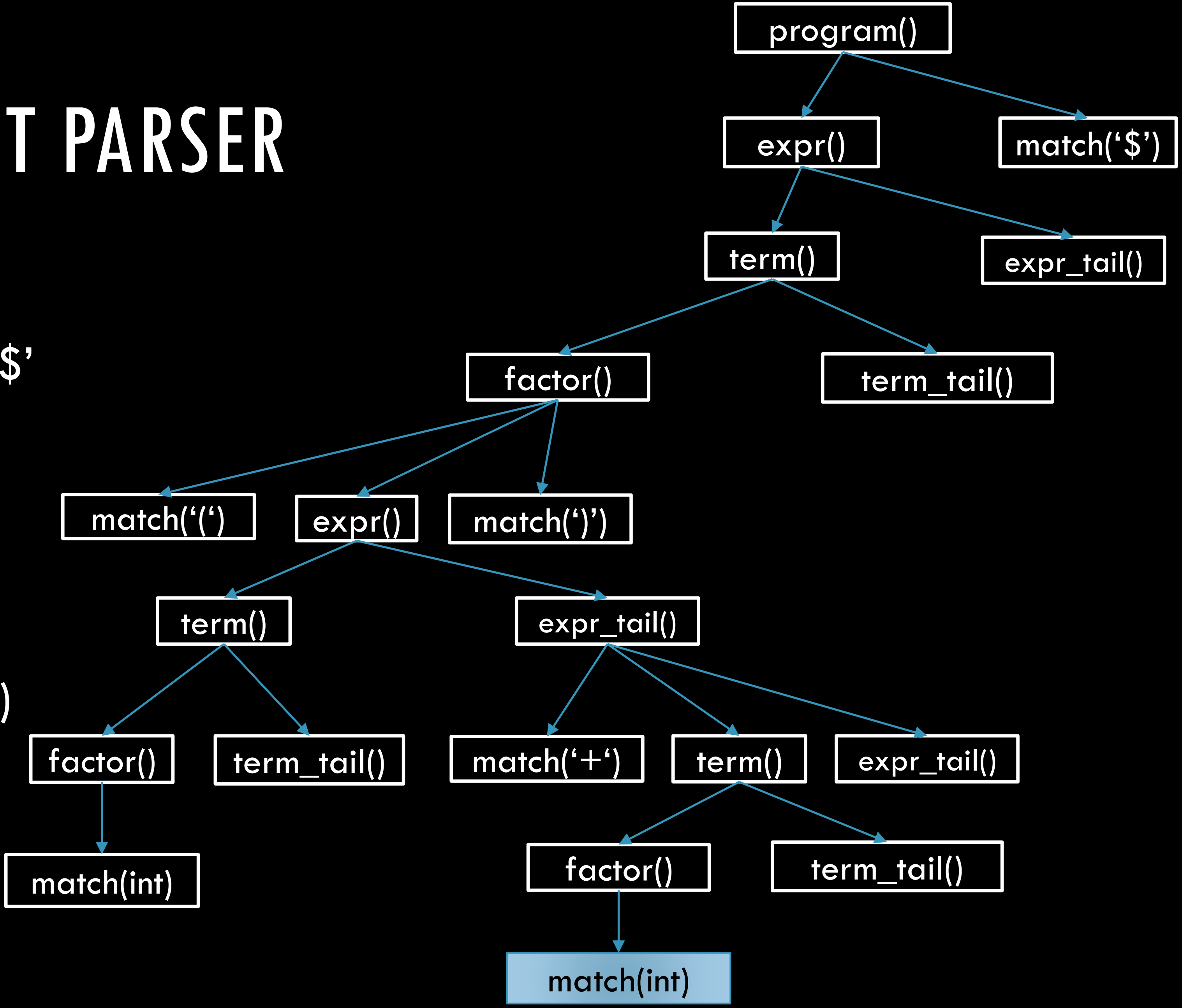
```
procedure factor
    case input of:
        '(': match('(') expr() match(')')
        int: match(int)
        else error
```



RECURSIVE DESCENT PARSER

int ‘)’ ‘*’ int ‘$’

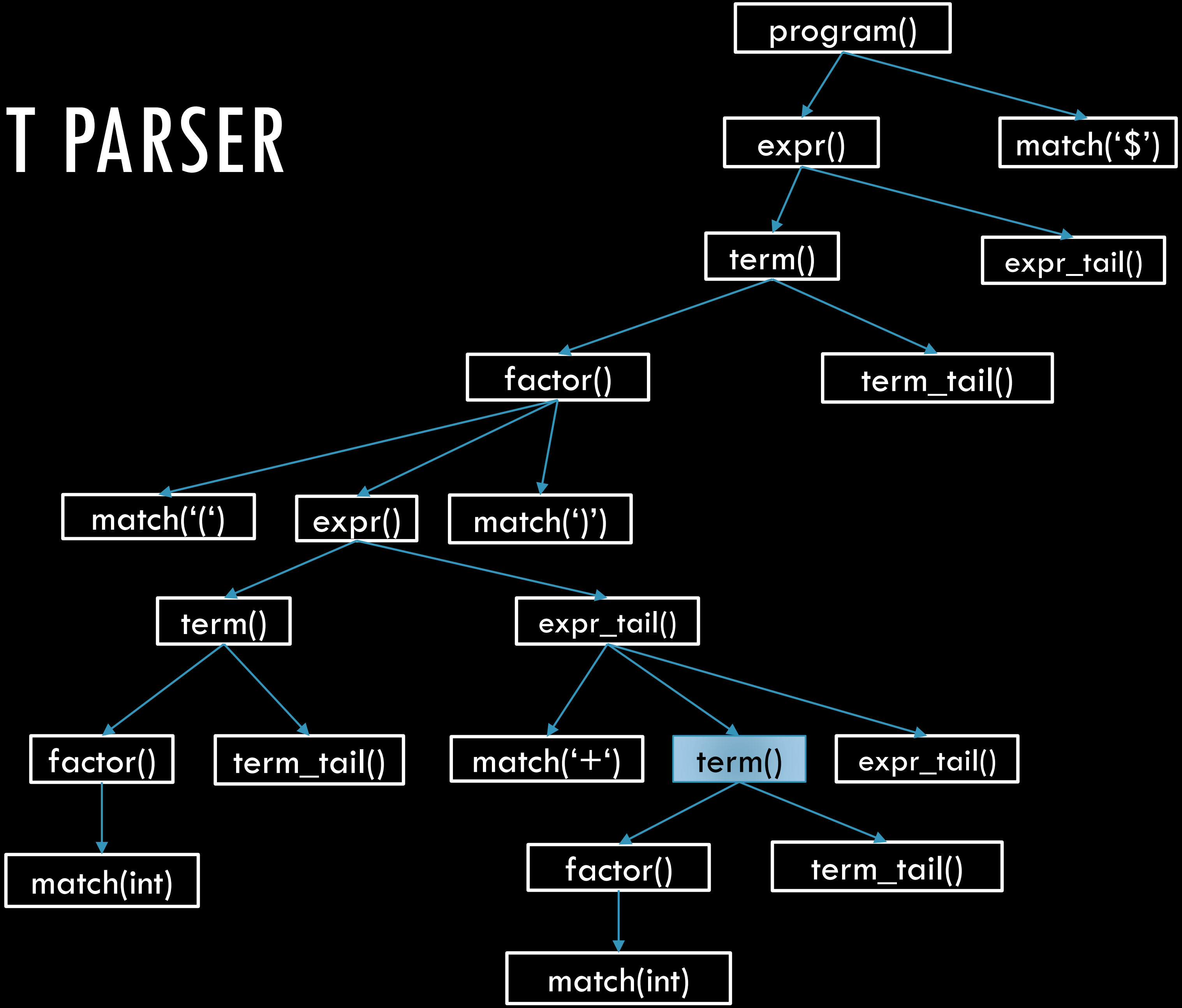
```
procedure factor
    case input of:
        '(': match('(') expr() match(')')
        int: match(int)
        else error
```



RECURSIVE DESCENT PARSER

‘)’ ‘*’ int ‘$’

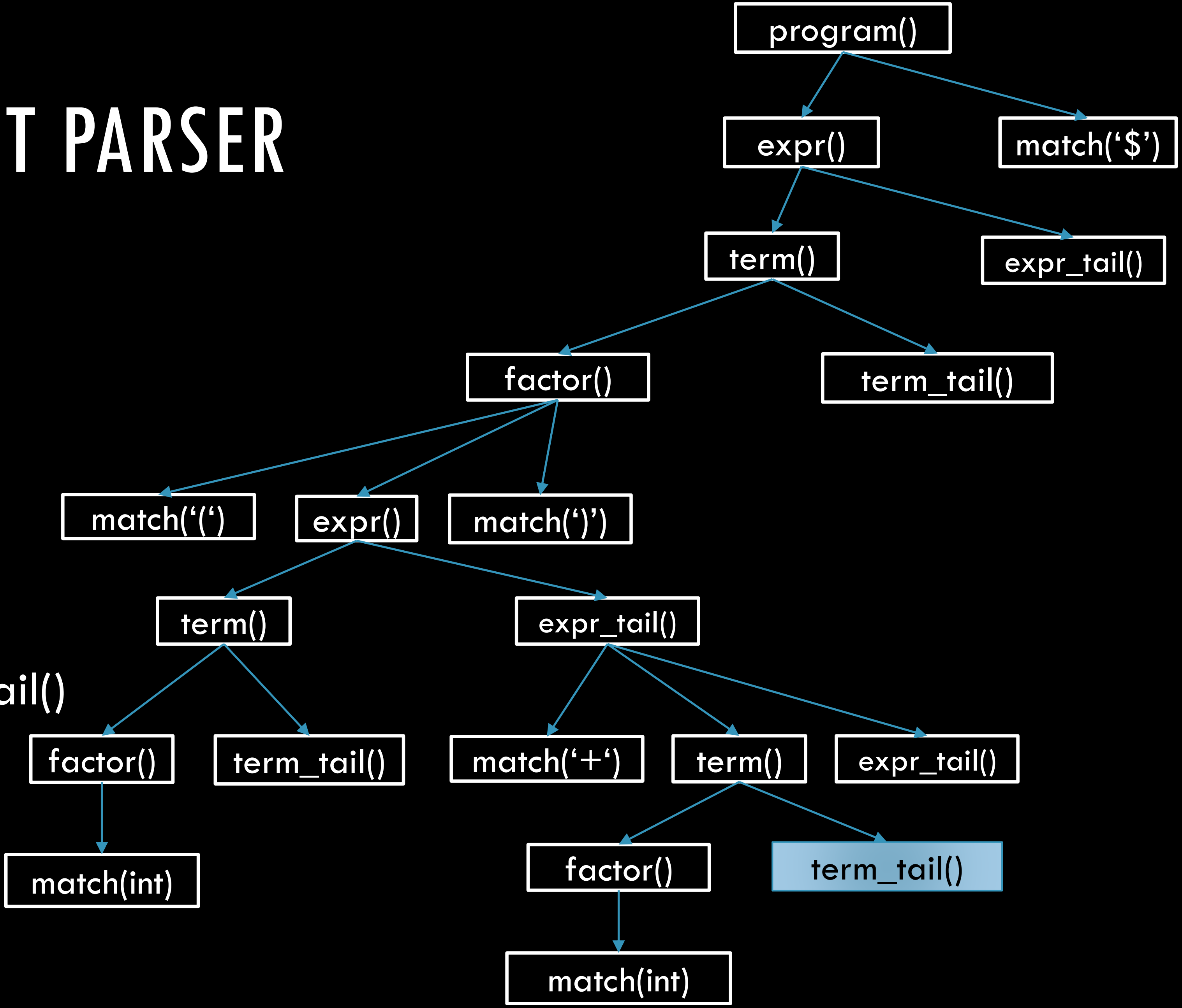
```
procedure term
    case input of:
        '(', int: factor() term_tail()
        else error
```



RECURSIVE DESCENT PARSER

‘)’ ‘*’ int ‘$’

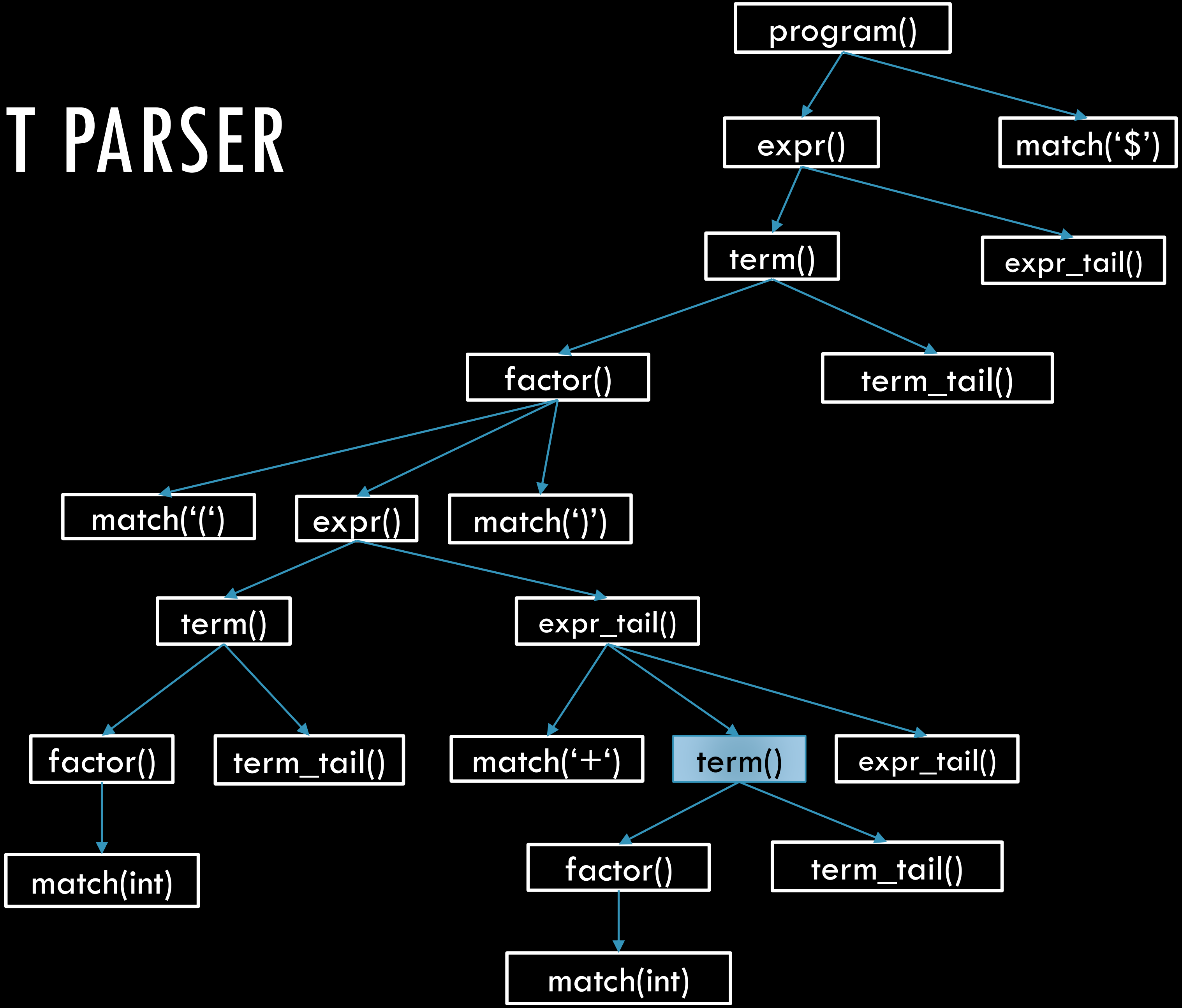
```
procedure term_tail
    case input of:
        '*': match('*') factor() term_tail()
        ‘)’, '+', '$': skip
        else error
```



RECURSIVE DESCENT PARSER

‘)’ ‘*’ int ‘$’

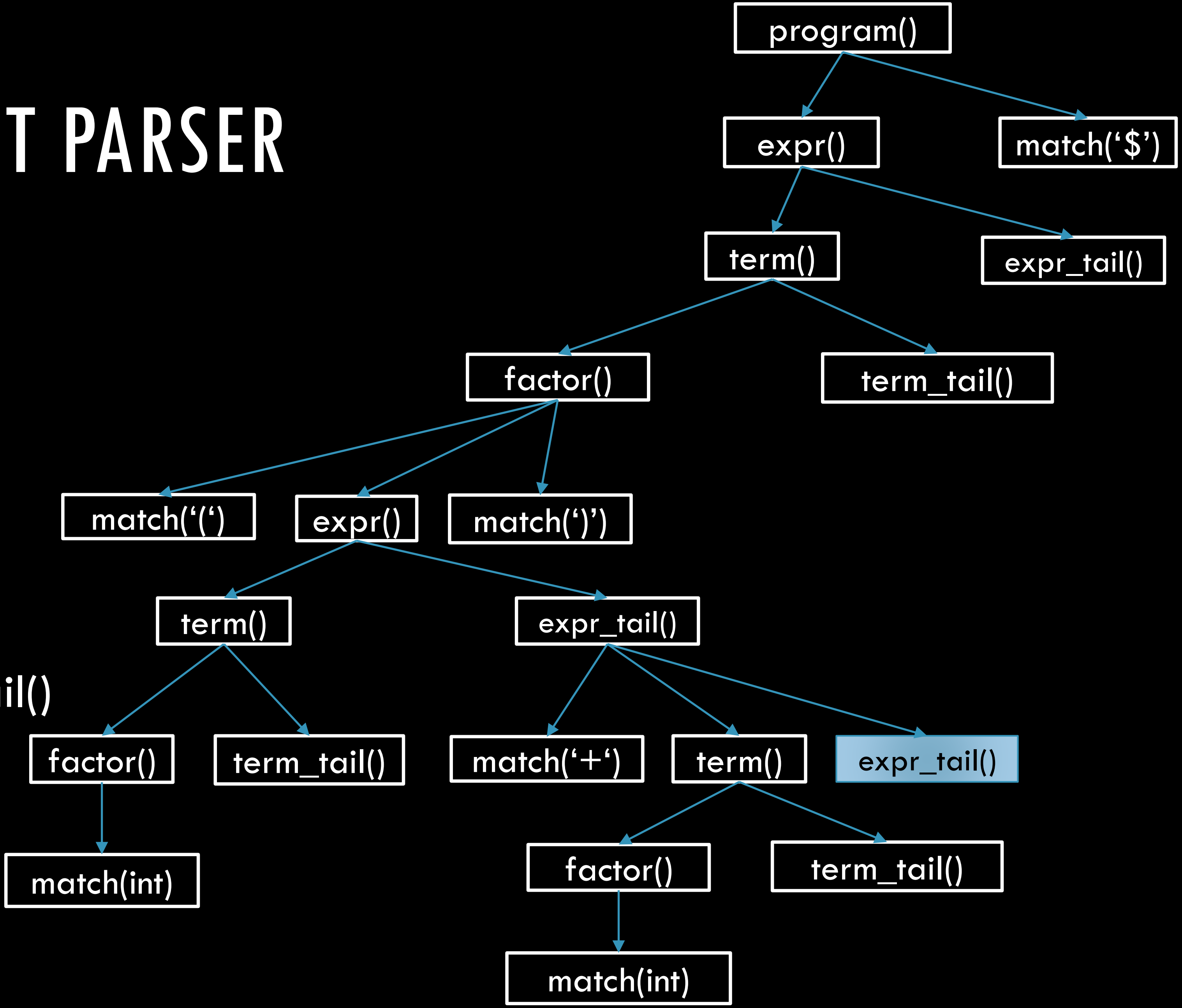
```
procedure term
    case input of:
        '(', int: factor() term_tail()
        else error
```



RECURSIVE DESCENT PARSER

‘)’ ‘*’ int ‘$’

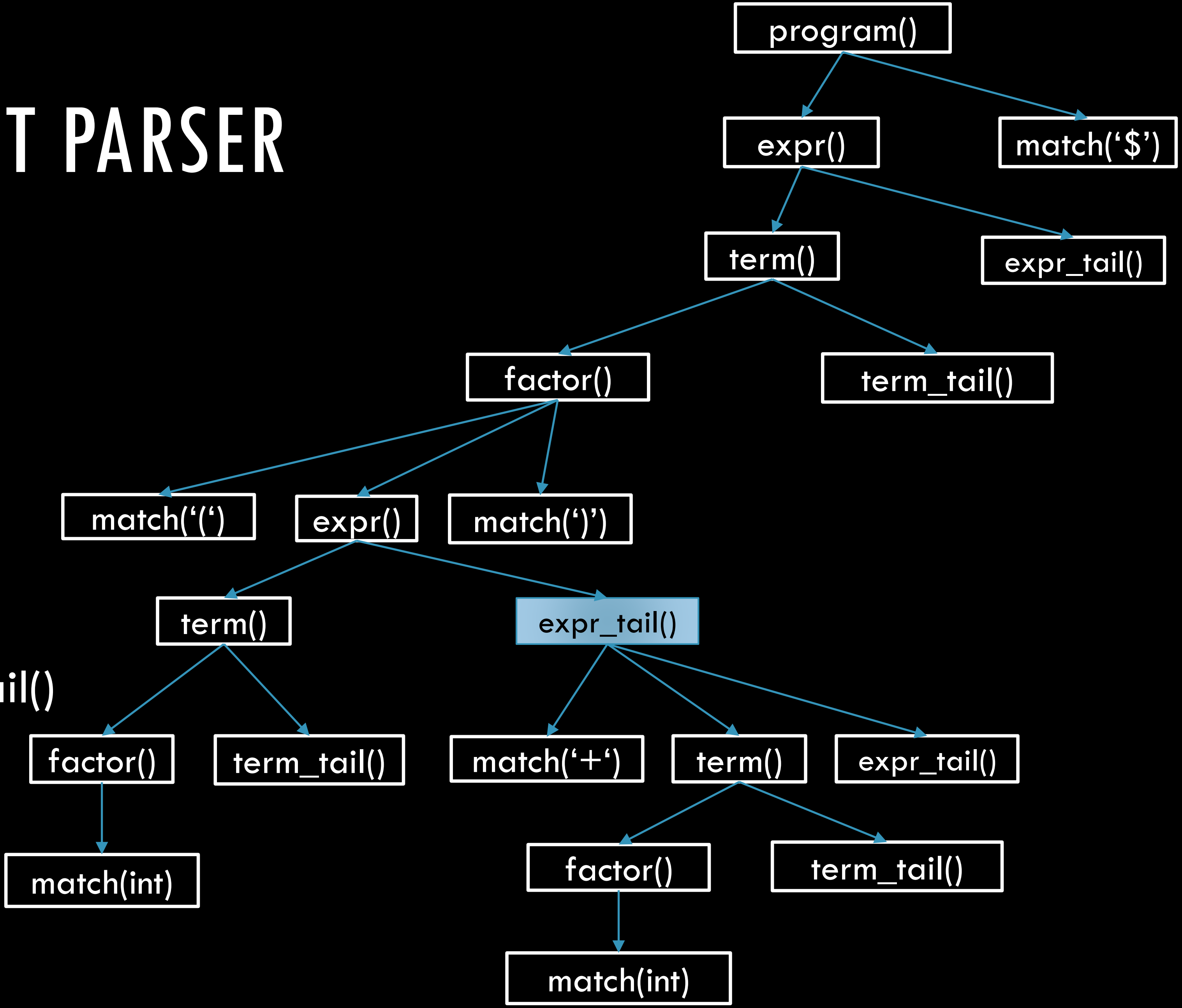
```
procedure expr_tail
    case input of:
        '+': match('+') term() expr_tail()
        ‘)’, '$': skip
        else error
```



RECURSIVE DESCENT PARSER

‘)’ ‘*’ int ‘$’

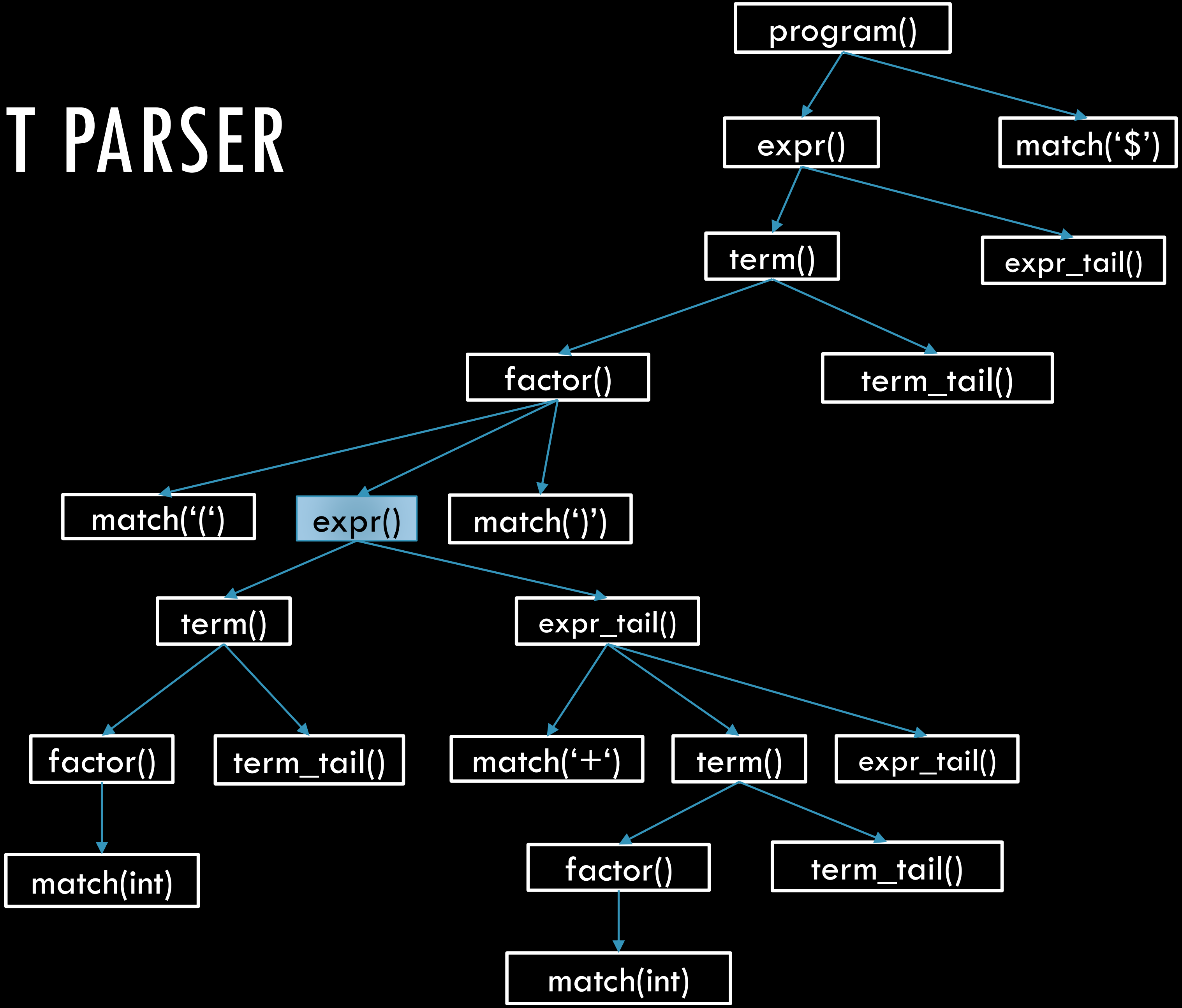
```
procedure expr_tail
    case input of:
        '+': match('+') term() expr_tail()
        ‘)’, '$': skip
        else error
```



RECURSIVE DESCENT PARSER

‘)’ ‘*’ int ‘$’

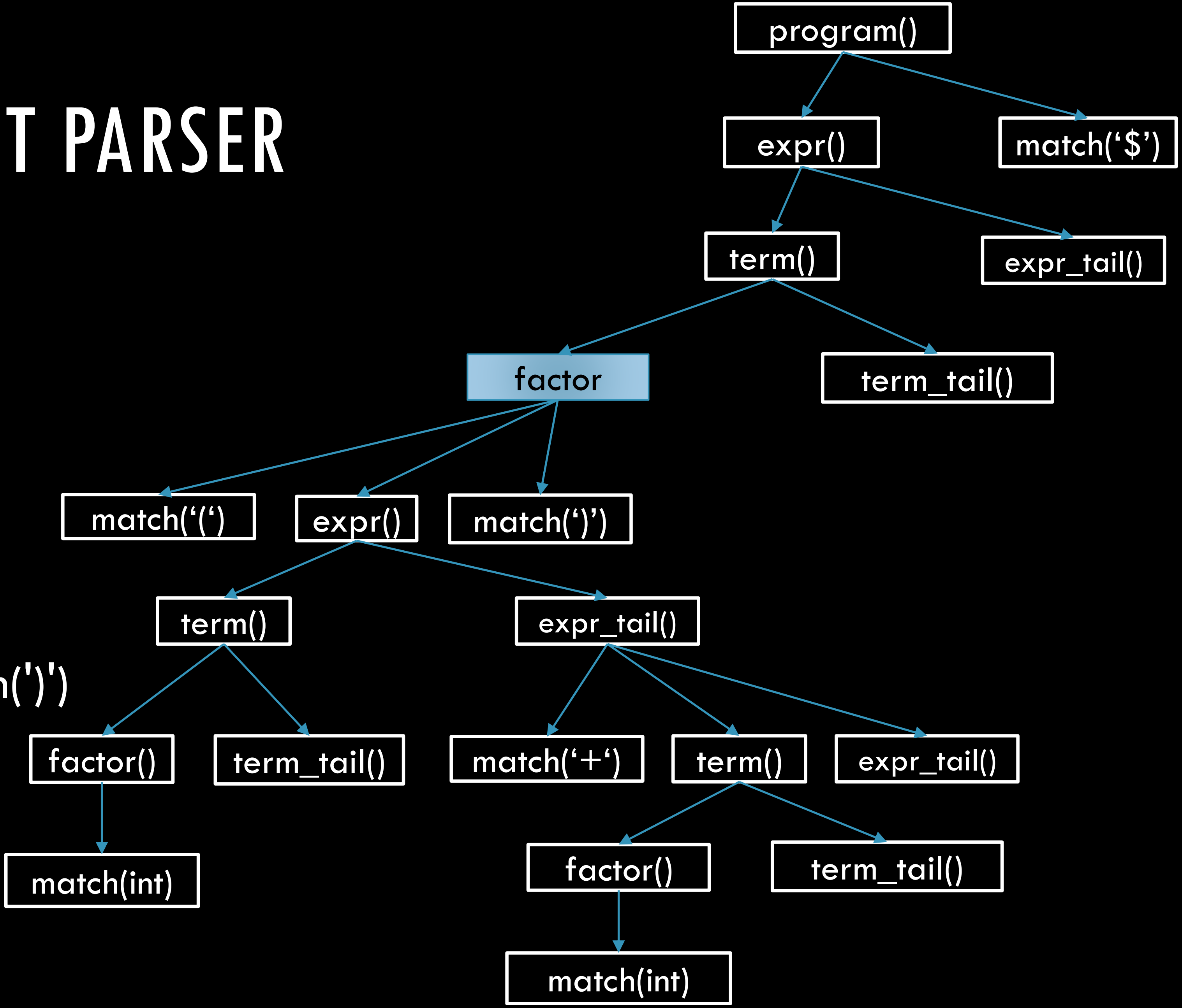
```
procedure expr
    case input of:
        '(', int: term() expr_tail()
        else error
```



RECURSIVE DESCENT PARSER

‘)’ ‘*’ int ‘$’

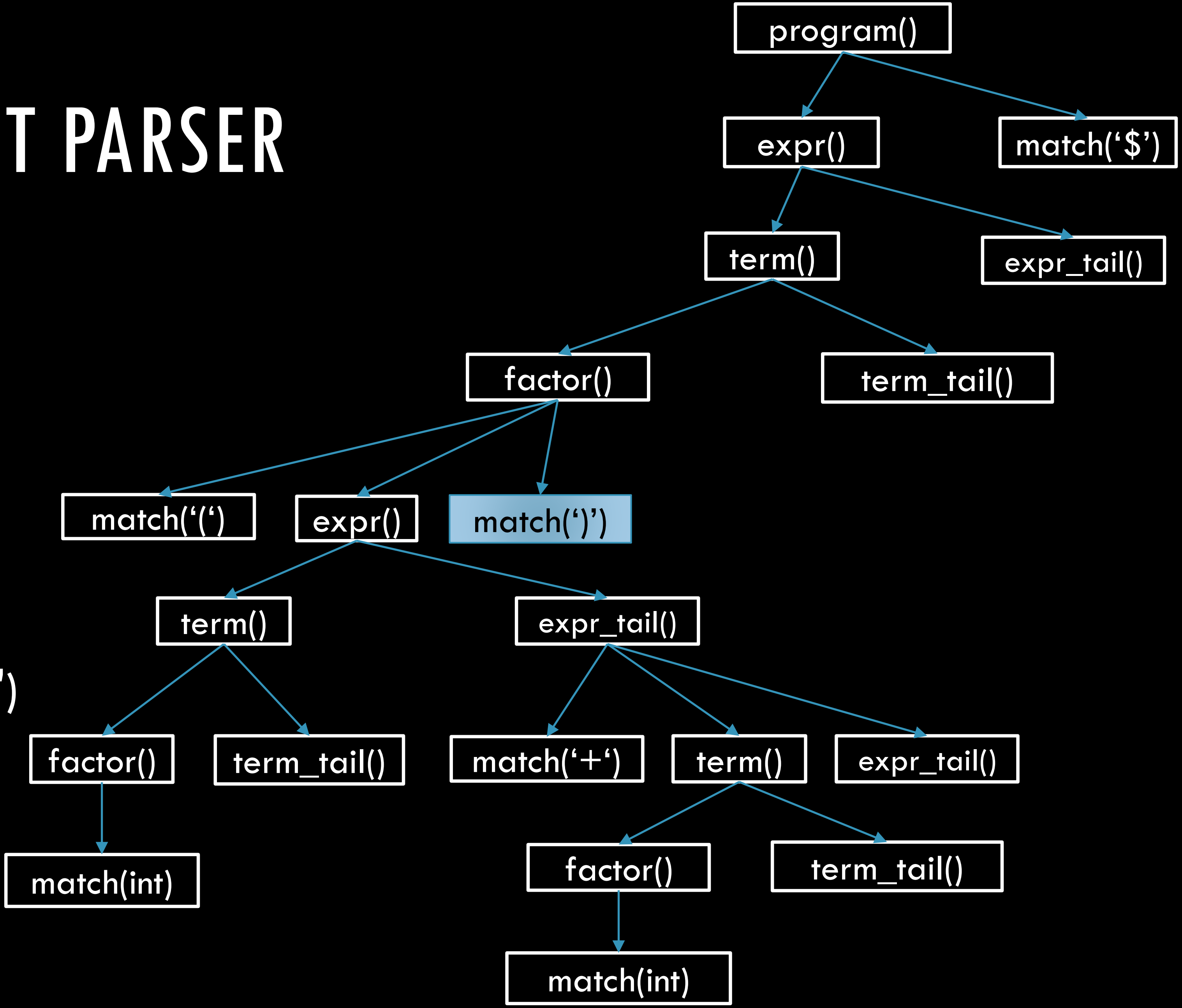
```
procedure factor
    case input of:
        '(': match('(') expr() match(')')
        int: match(int)
        else error
```



RECURSIVE DESCENT PARSER

‘)’ ‘*’ int ‘$’

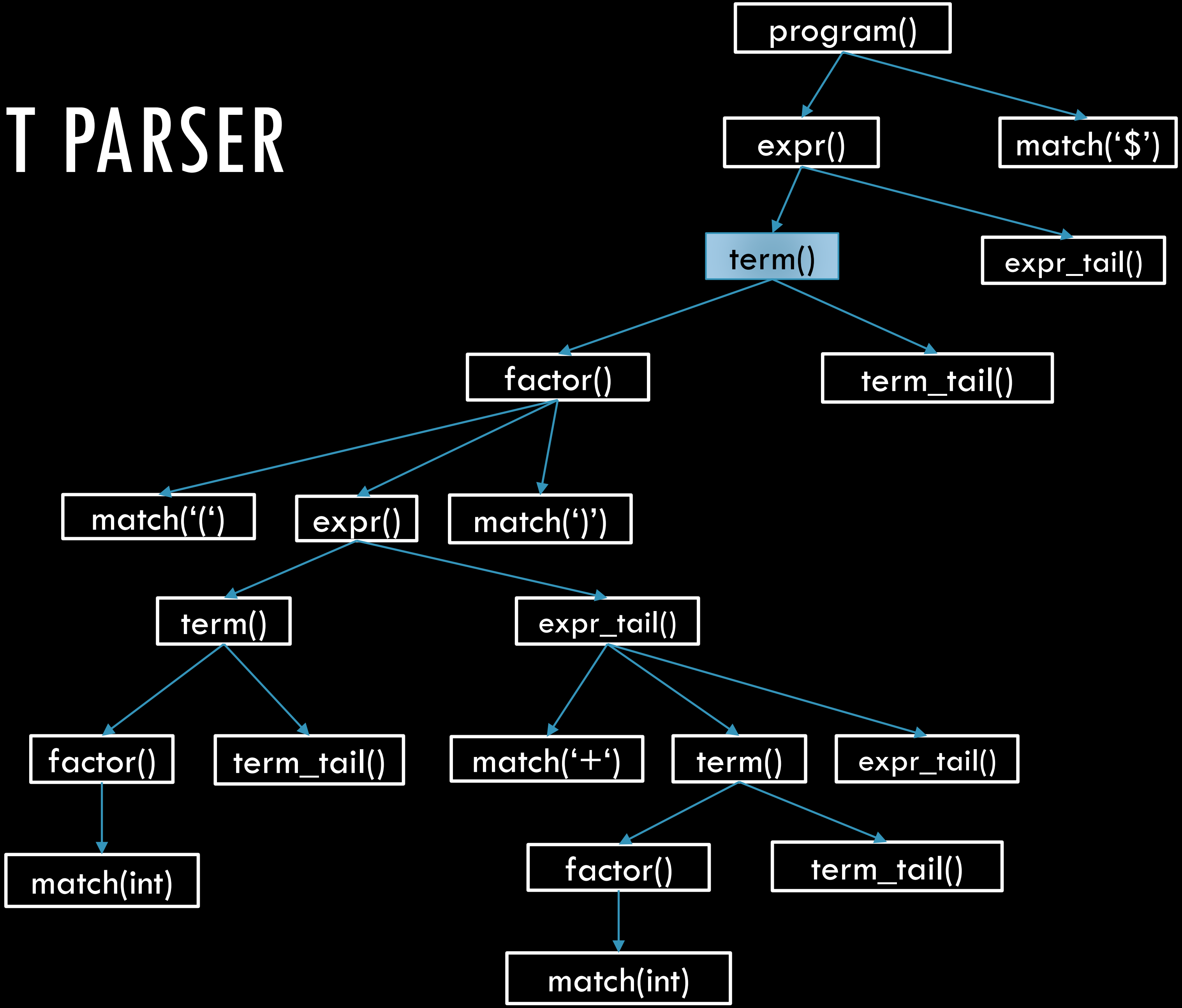
```
procedure factor
    case input of:
        '(': match('(') expr() match(')')
        int: match(int)
        else error
```



RECURSIVE DESCENT PARSER

‘*’ int ‘$’

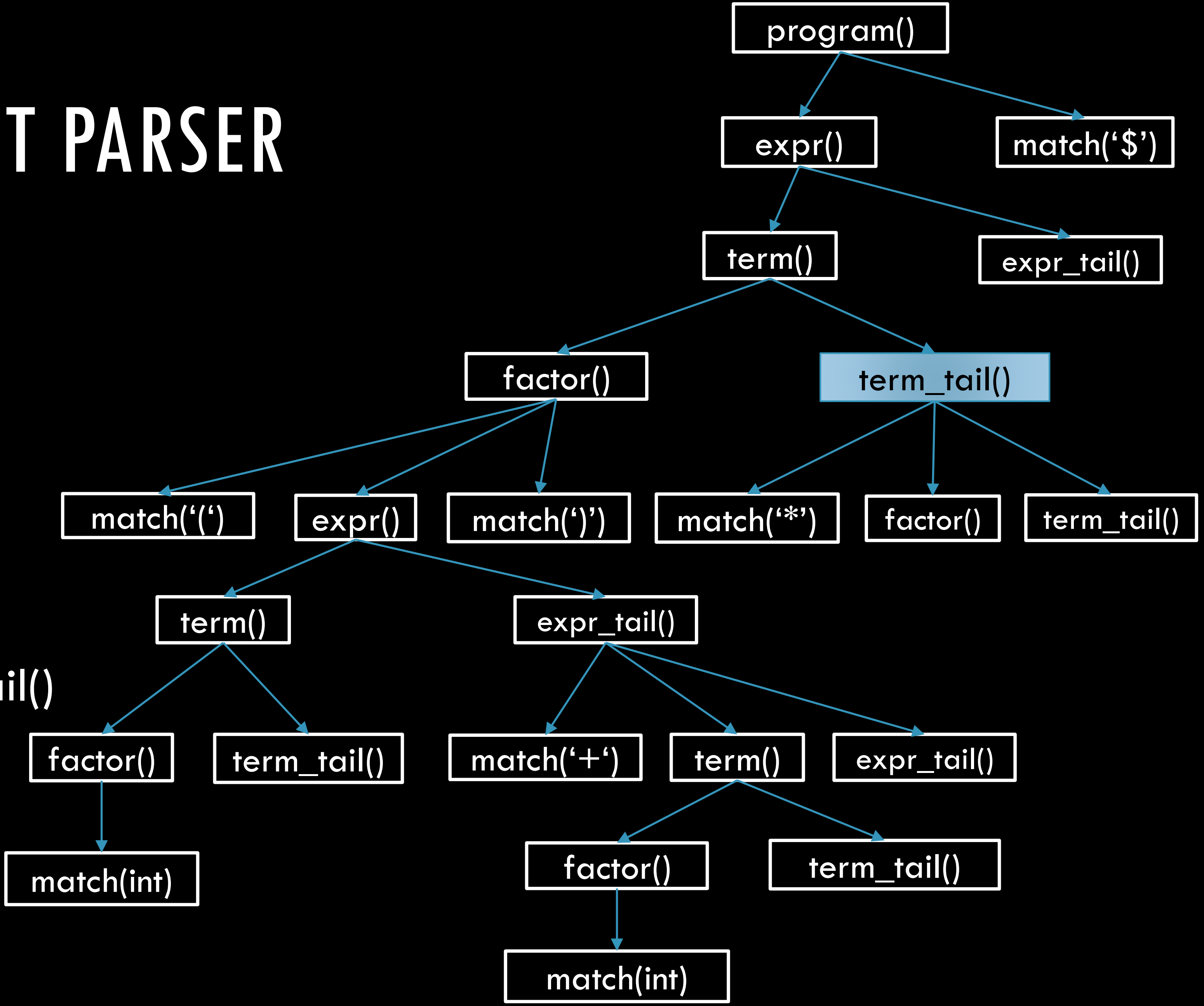
```
procedure term
    case input of:
        '(', int: factor() term_tail()
        else error
```



RECURSIVE DESCENT PARSER

‘*’ int ‘$’

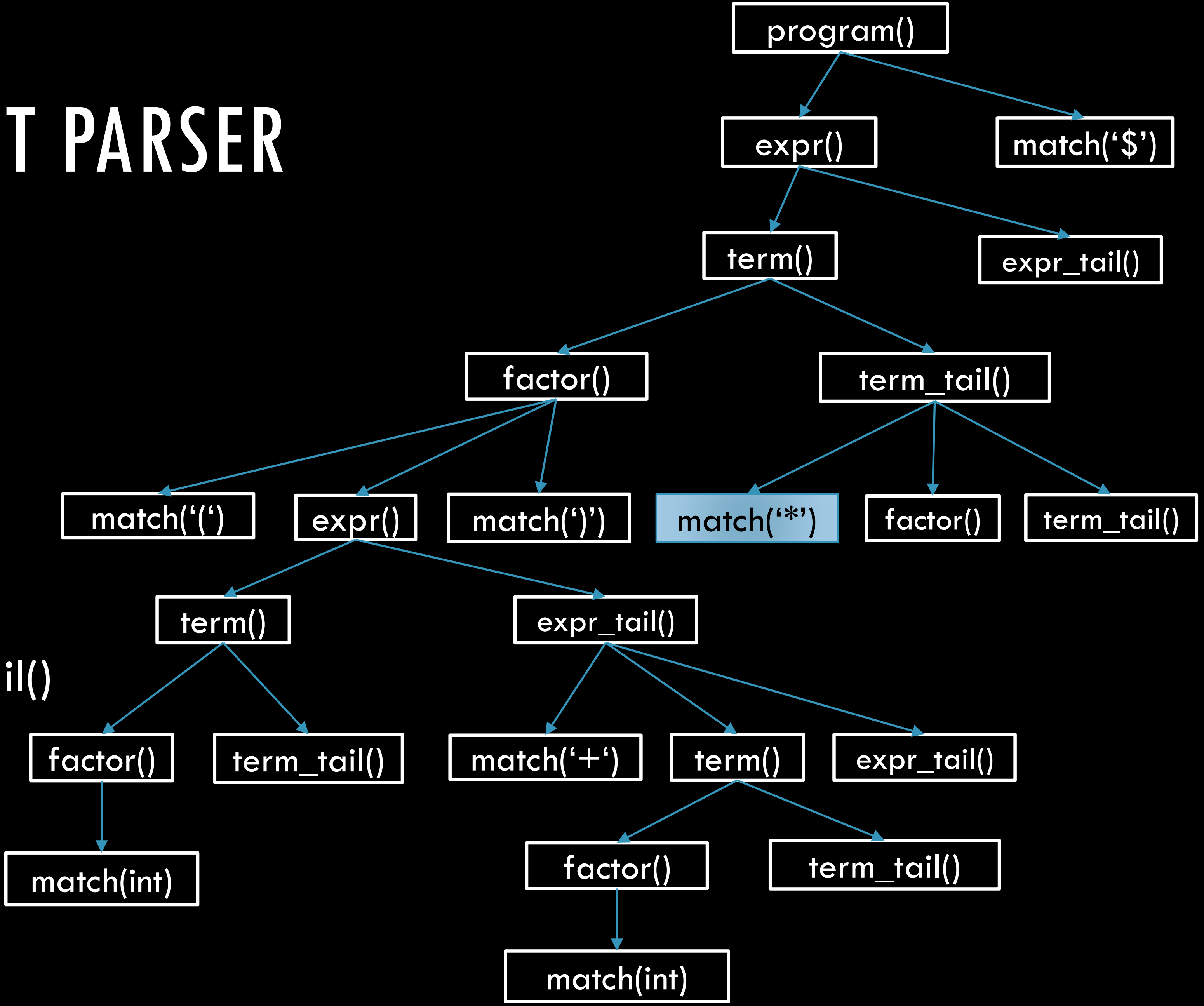
```
procedure term_tail
    case input of:
        '*': match('*') factor() term_tail()
        ‘)’, '+', '$': skip
        else error
```



RECURSIVE DESCENT PARSER

‘*’ int ‘$’

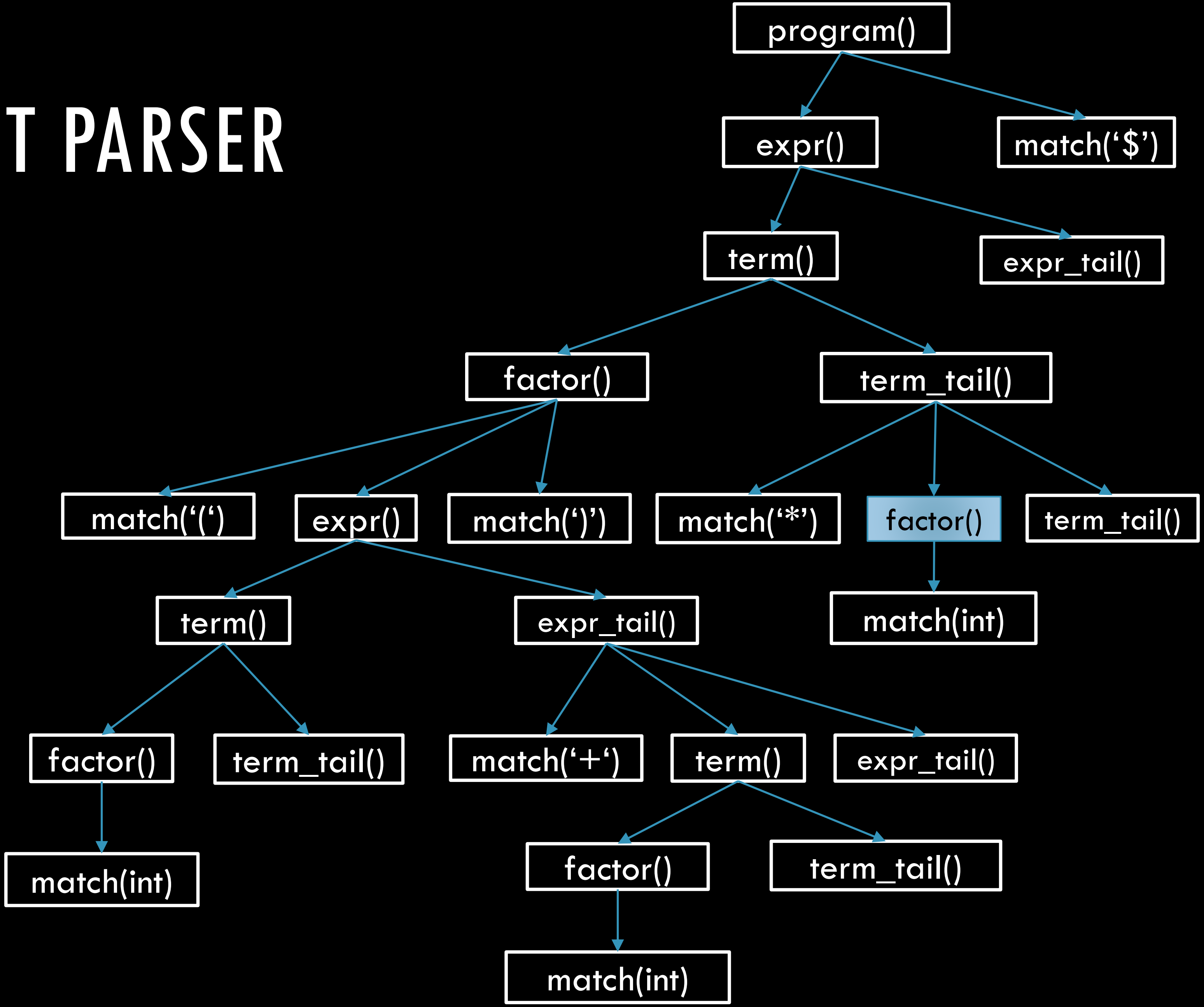
```
procedure term_tail
    case input of:
        '*': match('*') factor() term_tail()
        ‘)’, '+', '$': skip
        else error
```



RECURSIVE DESCENT PARSER

int ‘$’

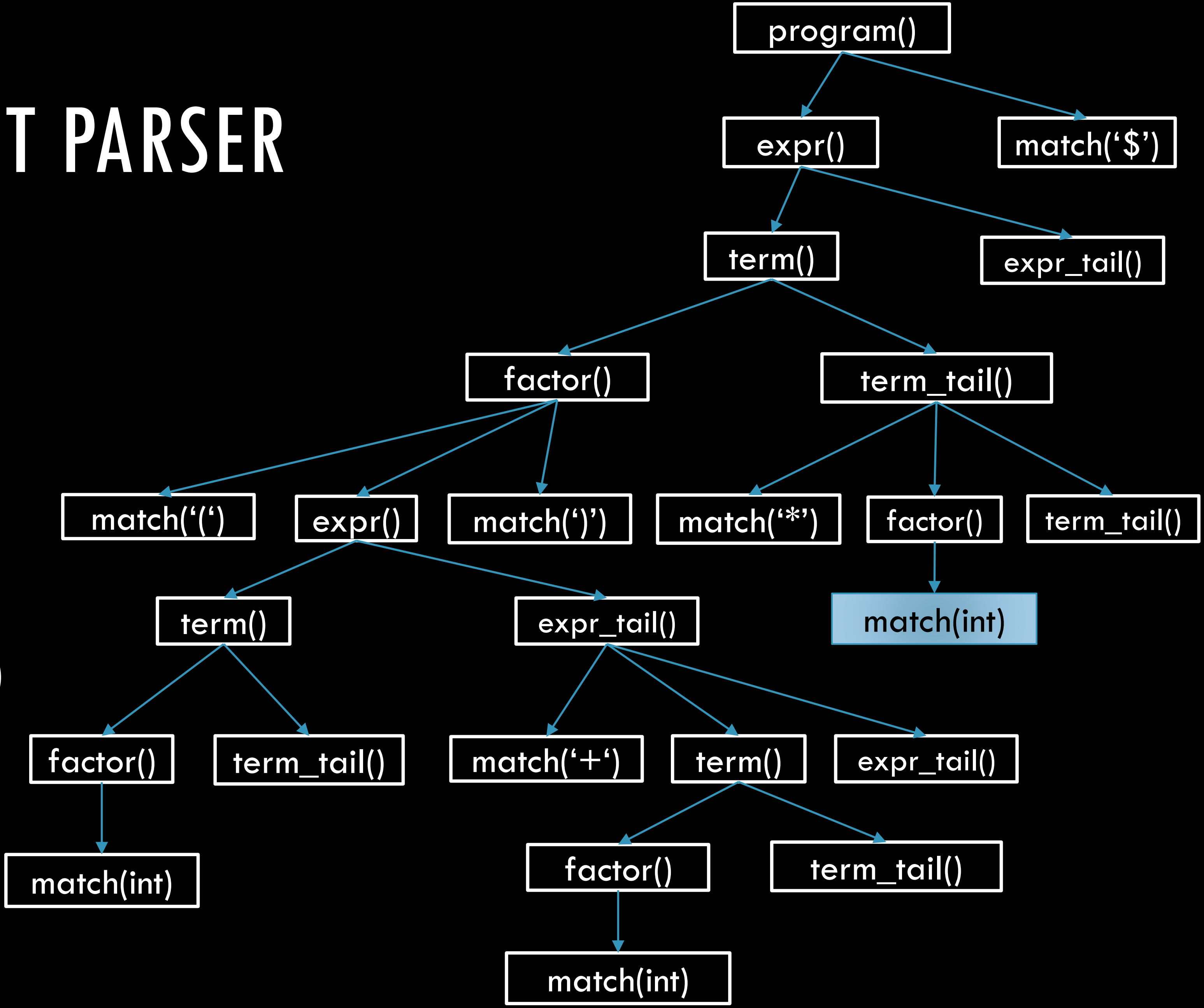
```
procedure factor
    case input of:
        '(': match('(') expr() match(')')
        int: match(int)
        else error
```



RECURSIVE DESCENT PARSER

int ‘$’

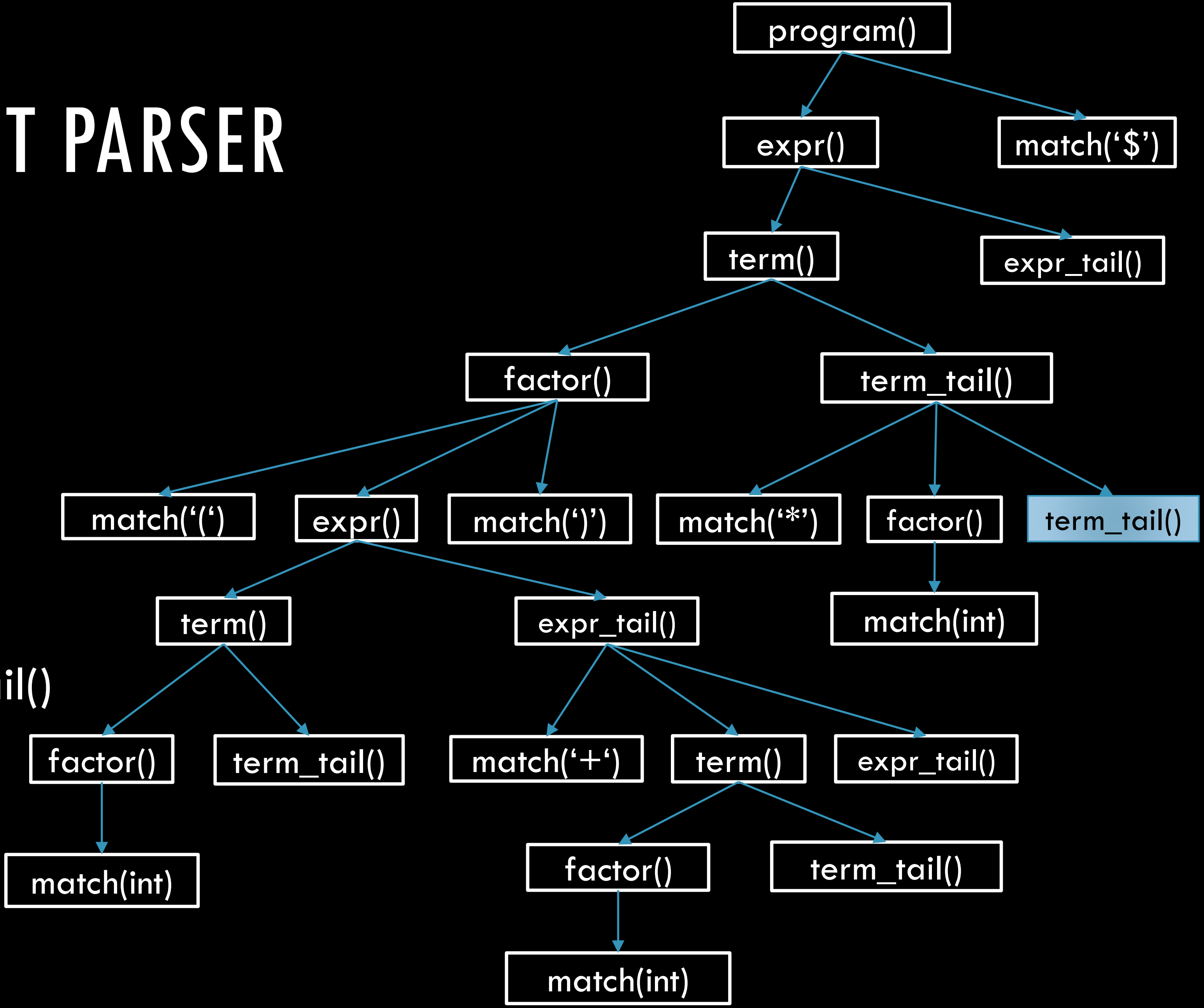
```
procedure factor
    case input of:
        '(': match('(') expr() match(')')
        int: match(int)
        else error
```



RECURSIVE DESCENT PARSER

‘$’

```
procedure term_tail
    case input of:
        '*': match('*') factor() term_tail()
        ‘)’, '+', '$': skip
        else error
```

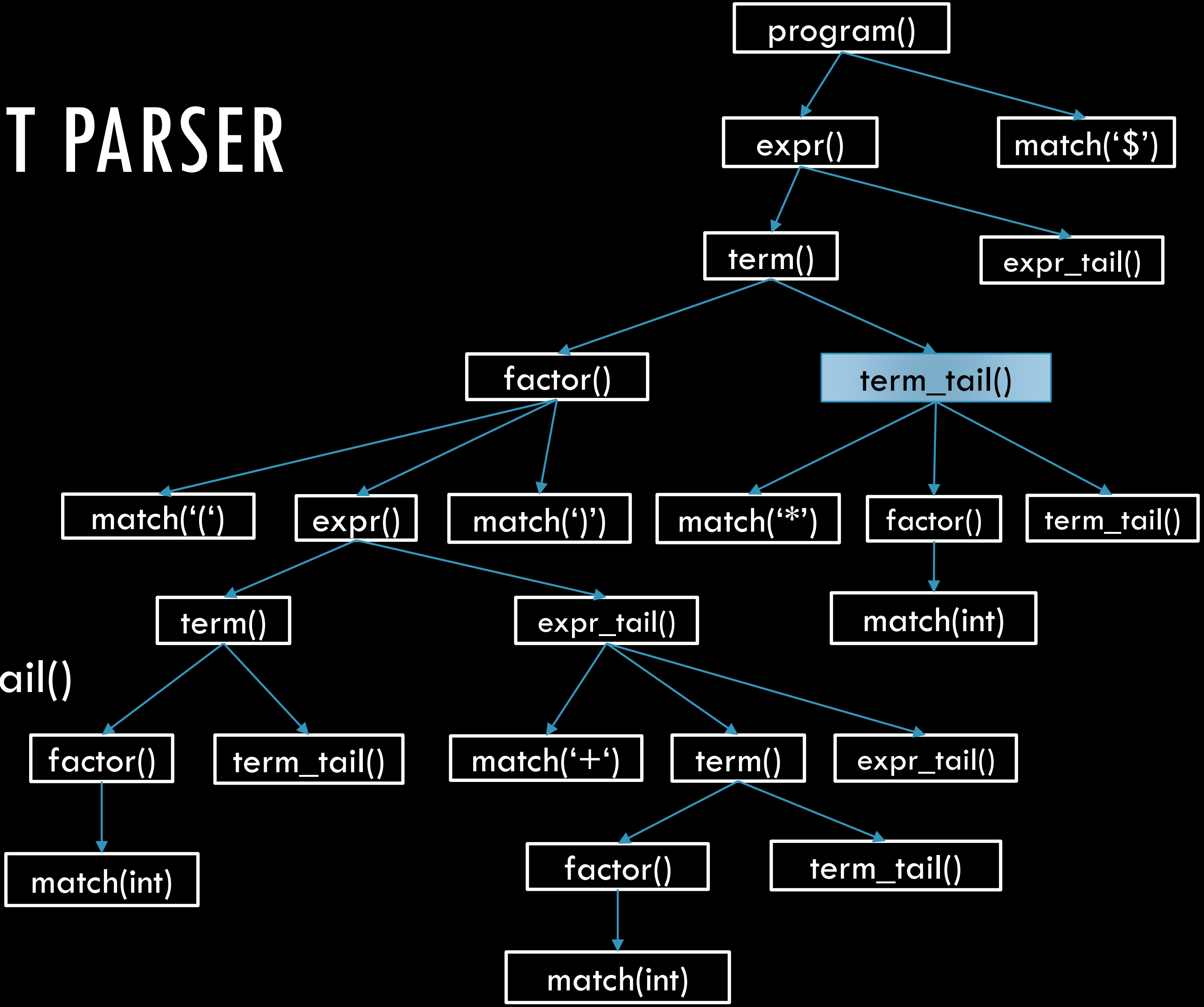


RECURSIVE DESCENT PARSER

‘$’

⬆

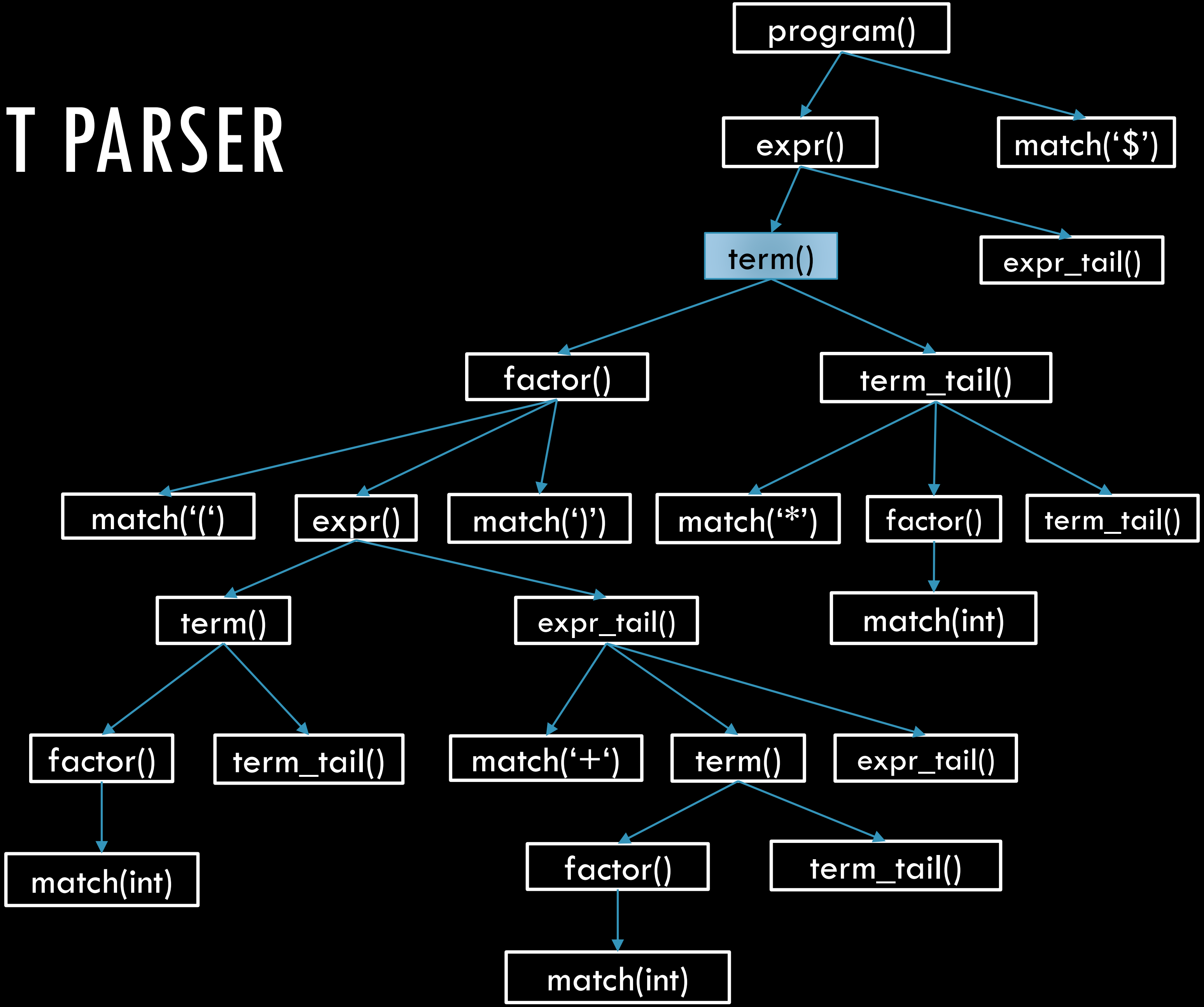
```
procedure term_tail
    case input of:
        '*': match('*') factor() term_tail()
        ‘)’, '+', '$': skip
        else error
```



RECURSIVE DESCENT PARSER

‘$’

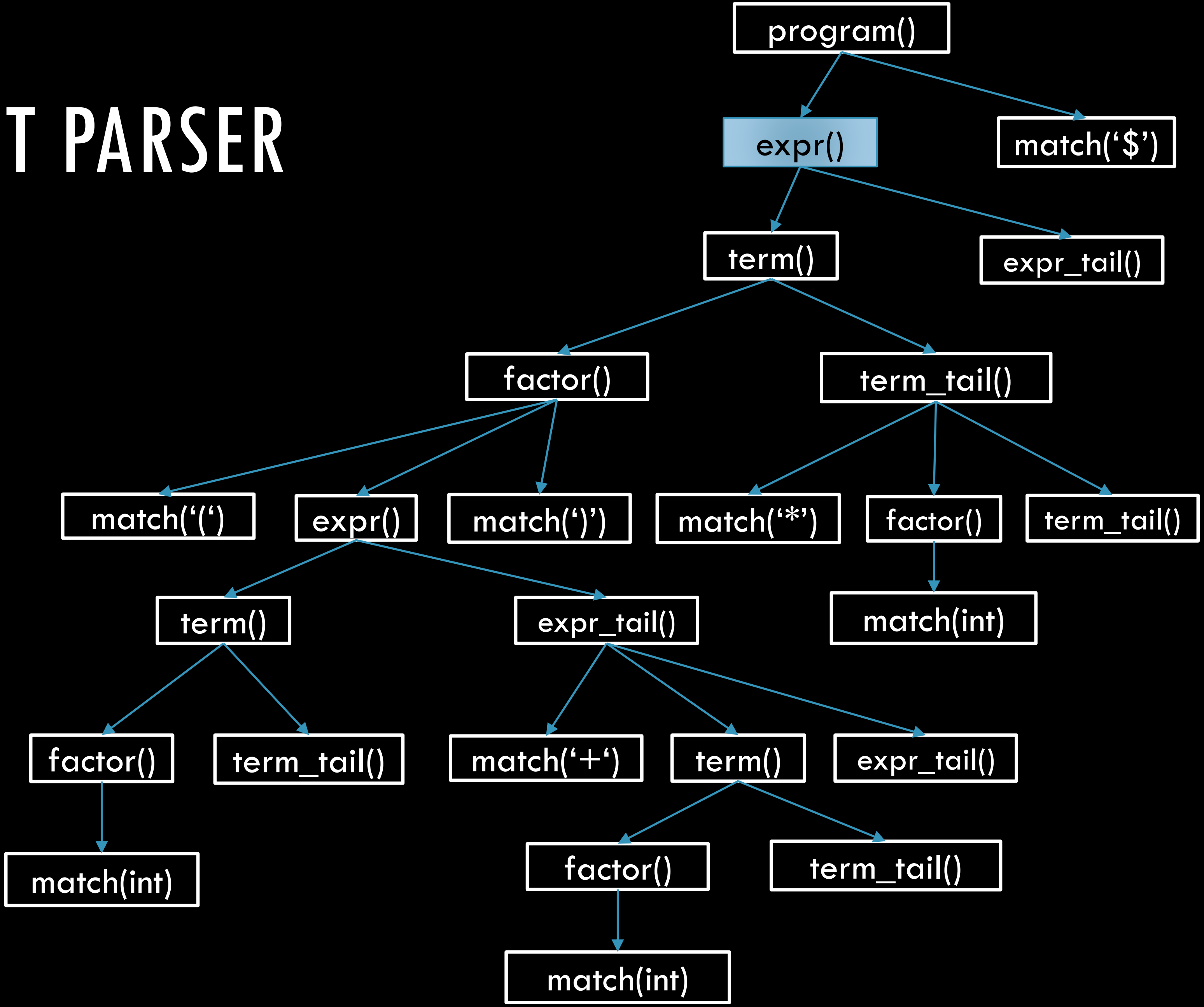
```
procedure term
    case input of:
        '(', int: factor() term_tail()
        else error
```



RECURSIVE DESCENT PARSER

‘$’

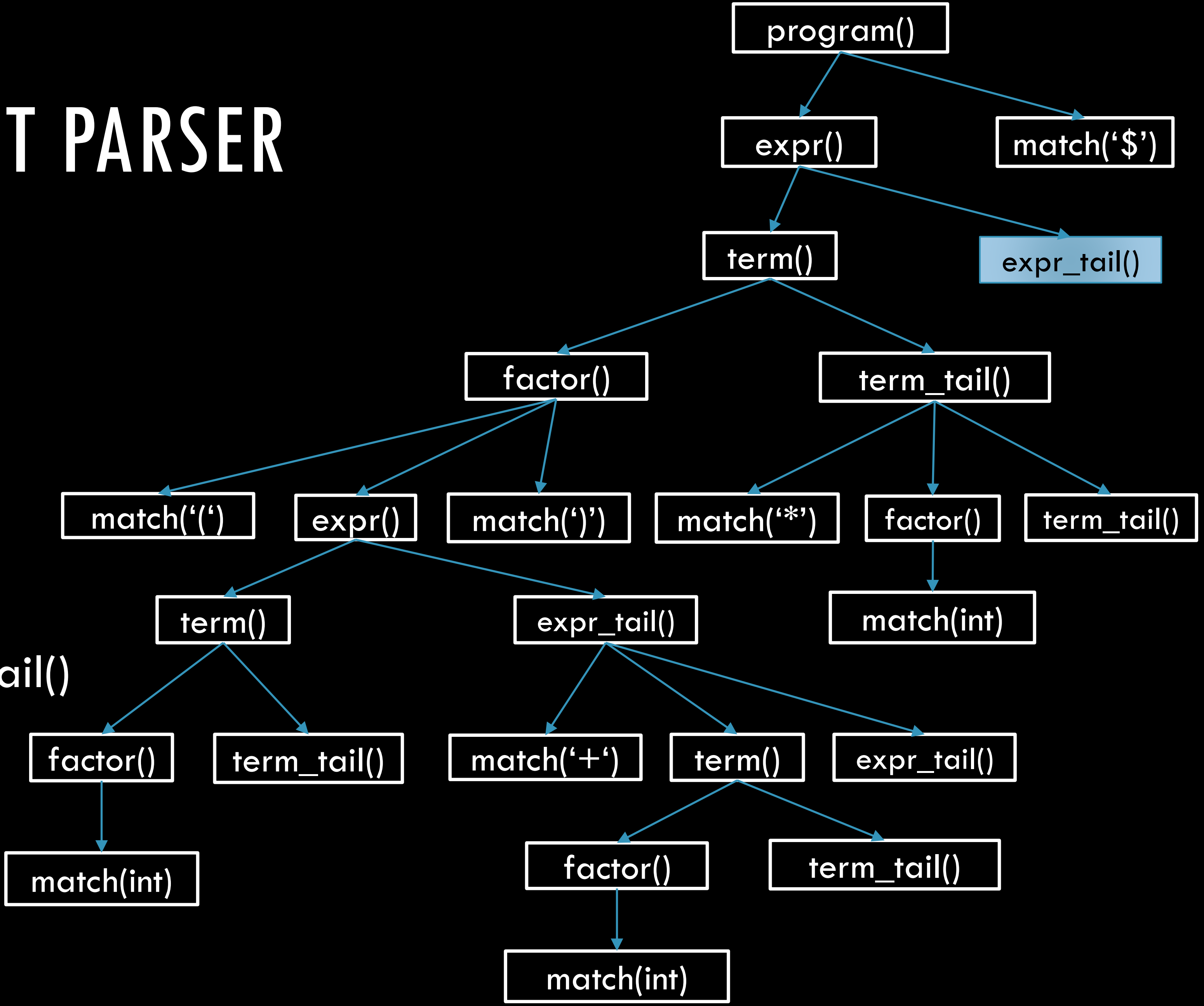
```
procedure expr
    case input of:
        '(', int: term() expr_tail()
        else error
```



RECURSIVE DESCENT PARSER

‘$’

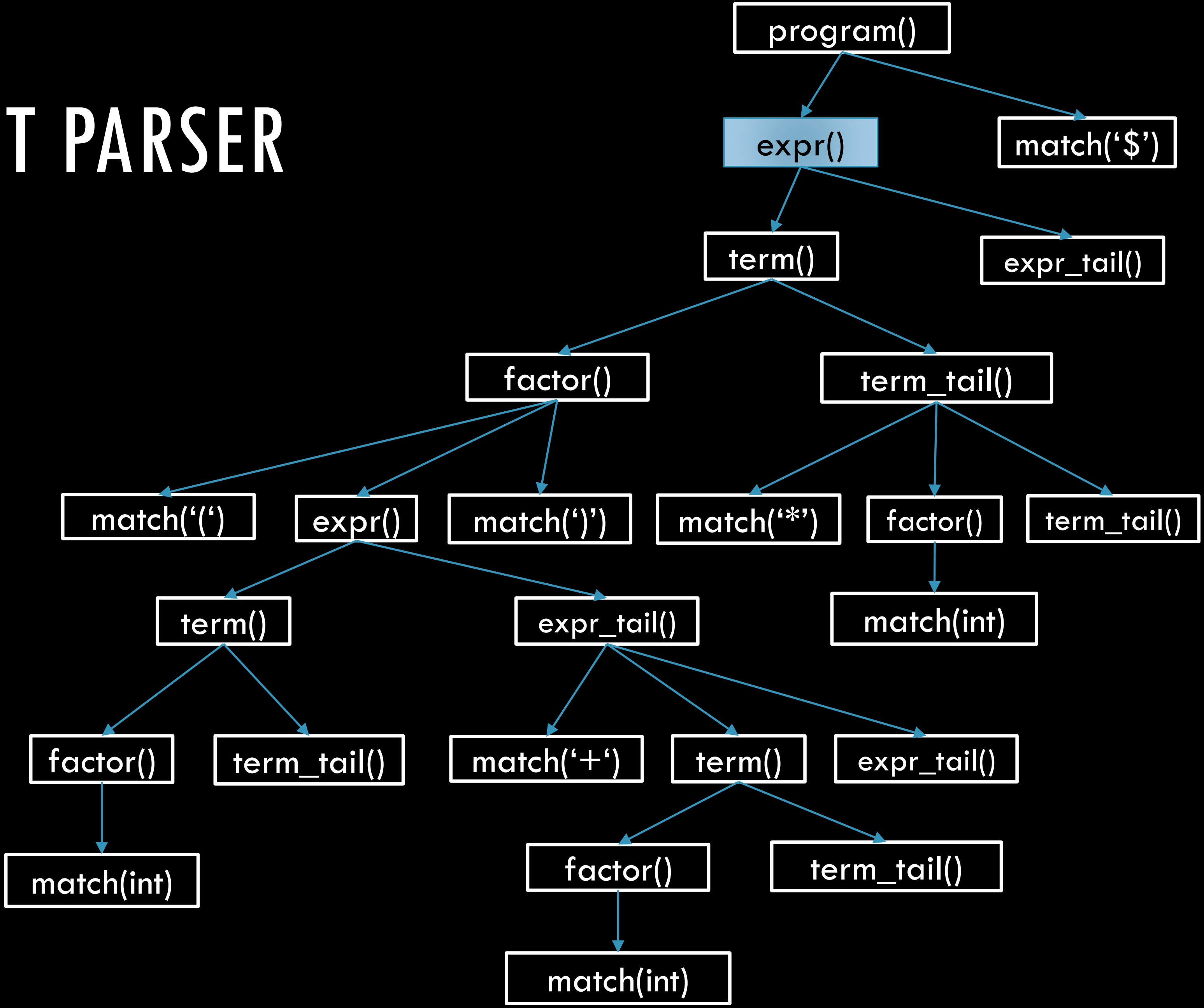
```
procedure expr_tail
    case input of:
        '+': match('+') term() expr_tail()
        ‘)’, '$': skip
        else error
```



RECURSIVE DESCENT PARSER

‘$’

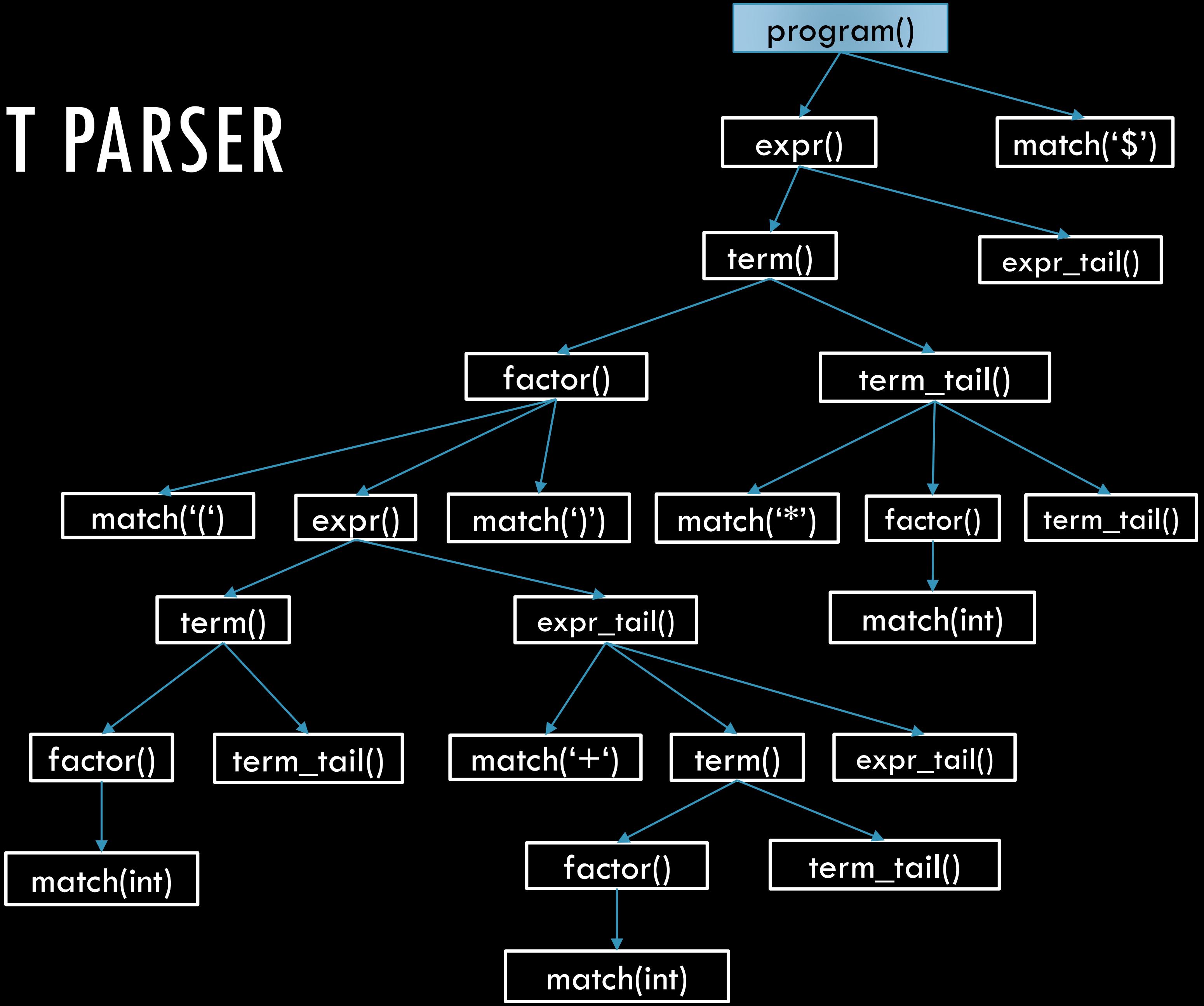
```
procedure expr
    case input of:
        '(', int: term() expr_tail()
        else error
```



RECURSIVE DESCENT PARSER

‘$’

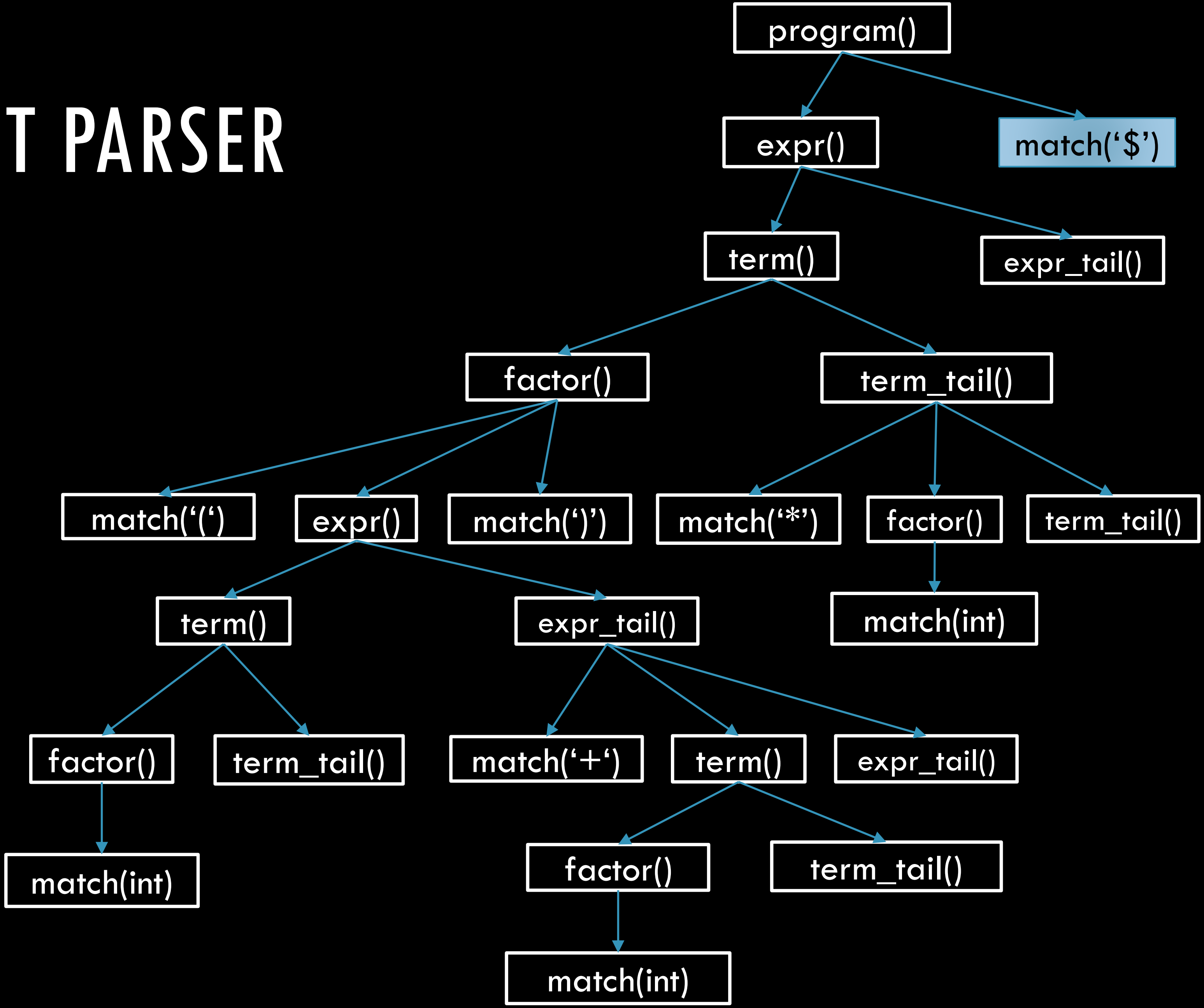
```
procedure program
    case input of:
        '(', int: expr() match('$')
        else error
```



RECURSIVE DESCENT PARSER

‘$’

```
procedure program
    case input of:
        '(', int: expr() match('$')
        else error
```



RECURSIVE DESCENT PARSER

We were able to process the entire string of tokens that the scanner generated for us without encountering an error. Therefore, our string is valid!

PARSE TREE

While parsing our string, we are also building a parse tree behind the scenes. The derivation we just performed via our recursive descent parser yields the following parse tree.

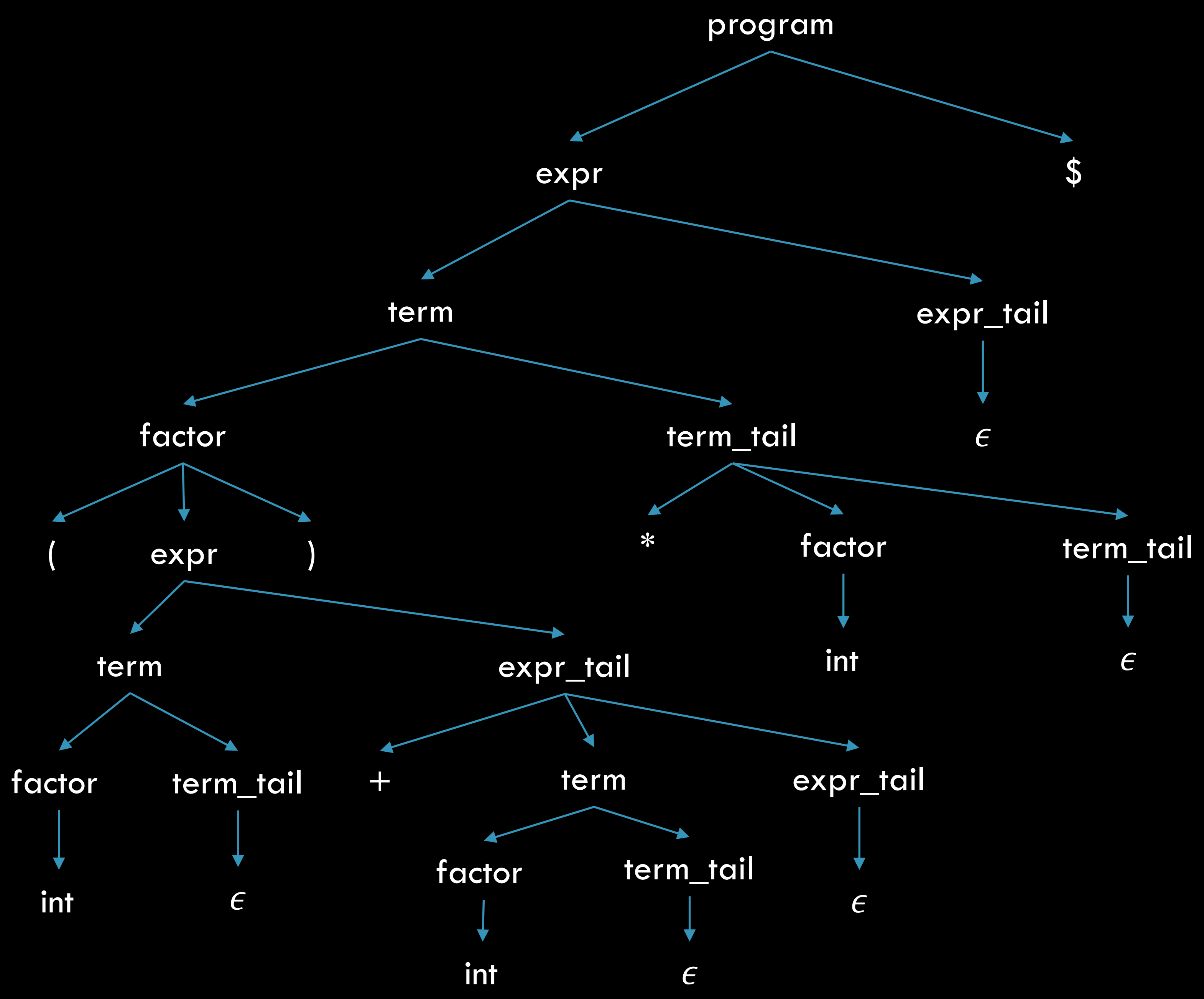


TABLE-DRIVEN TOP-DOWN PARSING

So, that’s a lot of work for a little parser! More complicated LL grammars necessitate a table-driven approach (as we need for our auto-generated scanners).

In a table-driven parser, we have two elements:

- A driver program, which maintains a stack of symbols. (language independent)
- A parsing table, typically automatically generated. (language dependent)

TABLE-DRIVEN TOP-DOWN PARSING

Here's the general method for performing table-driven parsing:

- We have a stack of grammar symbols. Initially, we just push the start symbol.
- We have a string of input tokens, ending with $.
- We have a parsing table M[N, T].
  - We can index into M using the current non-terminal at the top of the stack and the input token.

```
1. If top == input == '$': accept.
2. If top == input: pop the top of the stack, read new input token, goto 1.
3. If top is nonterminal:
        if M[N, T] is a production: pop top of stack and replace with production, goto 1.
        else error.
4. Else error.
```

TABLE-DRIVEN TOP-DOWN PARSING

Let's consider our grammar. First we need to create a table which reflects the possible productions based on the input.

(1) *program* → *expr*
(2) *expr* → *term expr_tail*
(3) *expr_tail* → + *term expr_tail*
(4) *expr_tail* → ϵ
(5) *term* → *factor term_tail*
(6) *term_tail* → * *factor term_tail*
(7) *term_tail* → ϵ
(8) *factor* → (*expr*)
(9) *factor* → int

N	int	(	)	+	*	$
program	(1)	(1)	-	-	-	-
expr	(2)	(2)	-	-	-	-
expr_tail	-	-	(4)	(3)	-	(4)
term	(5)	(5)	-	-	-	-
term_tail	-	-	(7)	(7)	(6)	(7)
factor	(9)	(8)	-	-	-	-

TABLE-DRIVEN TOP-DOWN PARSING

(1) *program* → *expr*
(2) *expr* → *term expr_tail*
(3) *expr_tail* → + *term expr_tail*
(4) *expr_tail* → ϵ
(5) *term* → *factor term_tail*
(6) *term_tail* → * *factor term_tail*
(7) *term_tail* → ϵ
(8) *factor* → (*expr*)
(9) *factor* → int

N	int	(	)	+	*	$
program	(1)	(1)	-	-	-	-
expr	(2)	(2)	-	-	-	-
expr_tail	-	-	(4)	(3)	-	(4)
term	(5)	(5)	-	-	-	-
term_tail	-	-	(7)	(7)	(6)	(7)
factor	(9)	(8)	-	-	-	-

Stack (bottom → top)	Input	Production Used
$ program	(int + int) * int $	
$ expr	(int + int) * int $	program → expr
$ expr_tail term	(int + int) * int $	expr → term expr_tail
$ expr_tail term_tail factor	(int + int) * int $	term → factor term_tail
$ expr_tail term_tail) expr (	(int + int) * int $	factor → (expr)
$ expr_tail term_tail) expr	int + int) * int $	
…	…	…

TABLE-DRIVEN TOP-DOWN PARSING

The algorithm is easy to follow, the key is to compute the parsing table. Basically, we need to make a choice of production for every pair of nonterminal and input characters.

So, when can *expr* be expanded with production *expr* → *term expr_tail*?

Intuitively, on any input token that can be the first token by expanding *term expr_tail*.

This should include all the first tokens obtained by expanding *term*, what are they?

Well, *term* → *factor term_tail* and *factor* → (*expr*) | int.

So our first tokens for expanding *expr* are '(' and int.

TABLE-DRIVEN TOP-DOWN PARSING

What if *term* could derive the empty string? We would also include the first tokens that can be derived from *expr_tail*. What if *expr_tail* could be the empty string too? Then we need to know what could follow *expr*.

Calculating an LL(1) parsing table includes calculating the *first* and *follow* sets. This is how we make decisions about which production to take based on the input.

FIRST SETS

Case 1: Let's say N → ω. To figure out which input tokens will allow us to replace N with ω, we calculate First(ω) – the set of tokens which could start the string ω.

- If X is a terminal symbol, First(X) = X.
- If X is ϵ, add ϵ to First(X).
- If X is a non-terminal, look at all productions where X is on left-hand side. Each production will be of the form: X → $Y_1 Y_2 \ldots Y_k$ where Y is a nonterminal or terminal. Then:
  - Put First(Y_1) - ϵ in First(X).
  - If ϵ is in First(Y_1), then put First(Y_2) - ϵ in First(X).
  - If ϵ is in First(Y_2), then put First(Y_3) - ϵ in First(X).
  - …
  - If ϵ is in Y_1, Y_2, …, Y_k, then add ϵ to First(X).

FIRST SETS

If we compute First(X) for every terminal and non-terminal X in a grammar, then we can compute First(ω), the tokens which can veritably start any string derived from ω.

Why do we care about the First(ω) sets? During parsing, suppose the top-of-stack symbol is nonterminal A and there are two productions A $\rightarrow$ α and A $\rightarrow$ β. Suppose also that the current token is a. Well, if First(α) includes a, then we can predict this will be the production taken.

FOLLOW SETS

Follow(N) gives us the set of terminal symbols that could follow the non-terminal symbol N. To calculate Follow(N), do the following:

- If N is the starting non-terminal, put EOF (or other program-ending symbol) in Follow(N).
- If X → αN, where α is some string of non-terminals and/or terminals, put Follow(X) in Follow(N).
- If X → $\alpha N \beta$ where α, β are some string of non-terminals and/or terminals, put First(β) in Follow(N). If First(β) includes ϵ, then put Follow(X) in Follow(N).

FOLLOW SETS

Why do we care about the Follow(N) sets? During parsing, suppose the top-of-stack symbol is nonterminal A and there are two productions A → α and A → β. Suppose also that the current token is a. What if neither First(α) nor First(β) contain a, but they contain ϵ? We use the Follow sets to determine which production to take.

NEXT LECTURE

Implementing an LL(1) parsing table.